TURING

图灵程序设计丛书

编程的原则

The Principles of Programming

改善代码质量的101个方法

[日] 上田勋◎著　支鹏浩◎译

初中级程序员高效进阶指南

加深技术理解　提高编程能力　优化团队合作

KISS /DRY/YAGNI/PIE/SLAP/OCP/10个软件架构基本技法/7个设计原理/无我编程/童子军规则/曳光弹/橡皮鸭调试法/破窗效应/80–10–10原则/第二系统综合征……


中国工信出版集团


人民邮电出版社
POSTS & TELECOM PRESS

上田勋

毕业于日本横滨国立大学经营学部。任职于Canon IT Solutions公司，从头参与了Web应用程序自动生成工具Web Performer的开发。在担任技术负责人、技术规范负责人、架构师和设计师的同时，自己也参与编程。喜欢读书，读过的技术书不少于800本，其运营的技术书读书博客上已有超过1500条博文。

封面设计 | 83720326@qq.com
自在书装设计

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

TURING

图灵程序设计丛书

编程的原则

The Principles of Programming

改善代码质量的101个方法

[日] 上田勋◎著　支鹏浩◎译

人民邮电出版社
北京

图书在版编目（CIP）数据

编程的原则：改善代码质量的101个方法 / (日) 上田勋著；支鹏浩译. -- 北京：人民邮电出版社，2020.6
（图灵程序设计丛书）
ISBN 978-7-115-53914-4

Ⅰ. ①编… Ⅱ. ①上… ②支… Ⅲ. ①软件质量 Ⅳ. ①TP311.5

中国版本图书馆CIP数据核字(2020)第073642号

内 容 提 要

本书介绍了软件开发领域101个重要的编程原则，涉及编程中的永恒真理，指导方针，编程思想，程序员的视角、习惯和工具，以及编程的反模式等内容。书中以“这个原则是什么”“为什么要遵循这个原则”“具体应该怎么做”为中心，对各个原则进行介绍，简明扼要，通俗易懂。这些原则凝聚了前人的智慧，经过了历史的考验，是指导程序员改善代码、进一步提升编程能力的实用指南。

本书适合各层次软件开发人员和项目管理人员阅读，也可作为高等院校计算机相关专业师生的参考读物。

◆ 著　　　[日] 上田勋
　译　　　支鹏浩
　责任编辑　杜晓静
　责任印制　周昇亮

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
　邮编　100164　　电子邮件　315@ptpress.com.cn
　网址　https://www.ptpress.com.cn
　北京　　　　　　印刷

◆ 开本：880×1230　1/32
　印张：10.25
　字数：306千字　　　　　　2020年6月第1版
　印数：1－3 000册　　　　 2020年6月北京第1次印刷

著作权合同登记号　图字：01-2017-2155号

定价：59.00元

读者服务热线：(010) 51095183转600　印装质量热线：(010) 81055316

反盗版热线：(010) 81055315

广告经营许可证：京东市监广登字20170147号




经验丰富的人，

往往只知其然而不知其所以然。

——海德格尔




前　言

不知道大家有没有过这样的经历：掌握编程语言的语法后会编写代码了，可写出的代码并不怎么好。

比如以下情况。

- 代码发布后频频发生故障
- 发布代码后收到“运行缓慢”“计算机出现宕机的情况”等投诉
- 让他人读完自己编写的代码后，收到“代码太难懂”“代码太复杂”“代码不够优雅”之类的评价
- 隔一段时间后再看自己编写的代码，会为其可读性之差而感到震惊
- 修改代码后，程序完全不能运行了
- 代码的修改给意料之外的地方造成了影响
- 想在代码中添加功能，却不知道该改动哪里
- 代码中没有添加功能的余地
- 修改说明文档后，相应部分的代码需要全部重写

当自己的编程能力到达一定程度之后，再继续提升就没有那么简单了。

如果能在一个合适的时间遇到一份好工作、一位好老师或一段好代码，那么我们就能取得进步。不过这得靠运气。好运砸中我们的概率并不大。

于是，很多人选择看书来提高自己的能力。不过，面向初学者的图书只能帮助我们掌握语法和一些基础知识，并不能帮助我们提高编程能力。而一些编程大师的著作对读者又有很高的要求。阅读这类图书容易让我们受挫，书中介绍的知识也不容易被我们理解。

况且，我们手头还有必须完成的工作，这些工作要求我们必须掌握

相关技术。于是，我们只能在工作的过程中一点一点提高自己的能力。

想成为一名更好的程序员，但苦于不知道该怎么做。前面的内容描述的就是这样一种状态。处于这种状态的人必不在少数。

笔者就是其中之一。于是，为了解决上述烦恼，笔者编写了本书。

* * *

本书介绍了软件领域非常有名的一些编程原则，这些原则能够帮助我们编写出优质的代码。笔者以“这个原则是什么”“为什么要遵循这个原则”“具体应该怎么做”为中心，对各个原则进行介绍。

作为编程指导方针的前提、准则、思想、习惯、视角、手法和法则等都属于编程的原则。这些原则经过了历史的考验，是帮助我们更好地进行编程的精髓知识。

之所以要学习编程的原则，是因为这些原则是帮助程序员成长的“捷径”。

编程的原则不是某种特定的技术，而是抽象度非常高的信息。因此，很多原则看上去让人有些摸不着头脑。不过，我们现在使用的各种技术，包括结构化编程、面向对象编程和函数式编程等，都是为了更好地实现编程原则的目的而诞生的。为了提高效率，这些编程技术还会不断进化。

编程原则的目的是解决编程中存在的本质问题，具体来说包括提高代码的可读性（降低代码的复杂度等）、降低编写代码的难度（减少代码量等）、避免出现 bug（消除编程的副作用等）、降低修改代码的难度（将修改带来的影响控制在一定范围），等等。为了实现编程原则的这些目的，诞生了很多技术，有的技术在诞生之后逐渐走向了消亡，只有那些可以真正解决本质问题的技术留存到了今天。

因此，只有理解了编程的原则，在学习具体技术时才能理解该技术存在的理由。这样一来，我们就能构建起知识网络，从而加快掌握技术的速度，加深对技术的理解。

另外，理解编程的原则后，我们在使用技术时也会变得更加熟练。编程的原则虽然不是一门具体的技术，但它能帮助我们学习和使用具体的技术。

编程的原则是一般化的信息，因此不会过时。无论使用什么样的技术，编程的原则都能指导我们编写代码，告诉我们应该采取什么样的行动。编程的原则是一经掌握就能一直为我们提供帮助的知识和智慧。

也就是说，编程的原则可以为程序员的成长打好基础，使程序员按照正确的方式取得进步，加速程序员的成长。

* * *

本书的目的是通过介绍编程的原则使读者写出优质的代码，成为一名优秀的程序员。

优秀的程序员会写出优质的代码。而优质的代码是可以进行改善的代码。可以进行改善的代码能造就更好的软件，这样的软件可以持续为用户提供服务。

希望本书有助于读者成为一名优秀的程序员。

上田勋

2016 年 1 月

注意事项

- 本书内容由作者独自研究完成。
- 如读者发现疏漏或错误，请与出版商联系。
- 对于使用本书内容的一切结果，本书作译者和出版社概不负责。
- 未经出版社书面许可，禁止复制本书的全部或部分内容。
- 本书中出现的公司名、商品名等一般是各公司的商标或注册商标。

目　录

序章　本书导读

第 1 章　前提　编程永恒的真理

第 2 章　准则　编程的指导方针



第 3 章　思想　编程的意识形态





第 4 章　视角　程序员的视角





第 5 章　习惯　程序员的日常





第 6 章　手法　程序员的工具箱





第 7 章　法则　编程的反模式






序章　本书导读

光有知识是不够的，还应当运用；
光有愿望是不够的，还应当行动。

——歌德






0.1 原则的分类

本书将编程的原则分为七章进行介绍，各章的内容及目标如下。

① 前提　编程永恒的真理

这一章介绍了编程中普遍存在的事实。

因为这部分介绍的原则与编程技巧无关，所以不常被人们提及。但这些原则是编程中至关重要且永恒不变的事实，是我们了解其他原则的前提。

我们要了解编程的本质，并以这些原则为前提来编写代码。

② 准则　编程的指导方针

这一章介绍了编程的指导方针。

其中的原则均是编程中基本的、适用于多种场合的“既定规则”。与具有绝对性的“铁律”不同，这些原则相对宽松，允许有例外出现。

因此，我们要按照这些原则的要求进行思考，并根据具体情况调整代码。这样一来，代码的质量自然会得到提升。

③ 思想　编程的意识形态

这一章介绍了编程思想方面的原则。

其中介绍的思想大致可分为编程理论、软件架构的基本技法、软件架构的非功能需求、七个设计原理、UNIX 思想和 UNIX 哲学这六类。各类思想都由多个原则构成。

成功的软件背后必然有文化、哲学和价值观等思想作为支撑。这些思想保证了软件的质量，使软件开发获得成功。

我们要学习这些在背后支撑着各个软件的思想，并将这些思想应用到编程的设计方针中。

④ 视角　程序员的视角

这一章介绍了编程中与视角、看法有关的原则。

擅长编程的人会为将来做打算，编写易于改善的代码。他们明白为了在将来省心，需要从哪种角度思考，兼顾哪些要素。所以说，特定的视角、看法能使代码“长寿”。

编程时我们往往要面临无数选择，这些视角将成为我们进行判断的依据。

⑤ 习惯　程序员的日常

这一章介绍了编程中与习惯有关的原则。

能写出优质代码的程序员必定养成了有利于写出优质代码的习惯，他们的行动方针也与写出优质代码息息相关。我们要了解他们日常所做、所想，并加以模仿。

⑥ 手法　程序员的工具箱

这一章介绍了编程中与手法、技术有关的原则。

编写代码是一种会给大脑带来负担的脑力劳动。如果不讲究方法、策略，大脑很快就会疲劳。这样一来，测试用例就可能存在遗漏，从而降低代码的质量，软件也会偏离用户的需求，成为废品。

为防止此类情况发生，编程高手往往会使用工具。这里说的工具可以是思想框架，也可以是设计手法或解决问题的方法。

我们要学会用这些工具来编写优质代码，开发好用的软件。

⑦ 法则　编程的反模式

这一章介绍了与软件开发中常见的陷阱有关的做法。

这些做法也叫作“反模式”，它们是从软件开发历史中归纳出来的反面教训，揭示了软件开发失败的原因。

提前了解软件开发中“做什么会导致什么”的因果关系，可以帮助我们在开发和运维中少走弯路。我们要对此有所了解。



0.2 介绍方式

笔者会按以下版块对各原则进行说明。

① 原则的名称

该版块用于介绍原则的名称，其中包括中文名和英文名，还有全称、略称和别名等。

② 是什么

该版块用于说明原则的含义。该部分只对原则内容进行简单说明，复杂的内容以及应用方面的内容将在“扩展”版块中介绍。

③ 为什么

该版块用于说明原则的必要性。这是本书最重要的部分。明白了使用原则的目的之后，我们自然就能想到实现目的的手段（技术）了。

④ 怎么做

该版块用于介绍原则的运用方法。其中会介绍有助于灵活运用原则的思考方针，还会介绍将方针与原则联系起来的方法。

⑤ 扩展

该版块用于介绍原则的延伸内容。

⑥ 关联信息

该版块用于介绍与该原则相关的、对编程有益的信息。

⑦ 出处

该版块用于介绍原则的主要来源。此处选取的是最具参考价值的图

书，不过这些图书不一定是该原则最早出现的地方。

⑧ 相关图书

该版块用于介绍包含了原则的相关信息或延伸信息的图书。

①

2.6 OCP

英 语 Open-Closed Principle

中 文 开闭原则

② 是什么 代码的修改不相互影响

我们要让代码同时满足对扩展开放、对修改关闭这两个属性。

对扩展开放表示代码的行为可以扩展。

对修改关闭表示当对代码的行为进行扩展时，其他代码完全不会受到影响。

代码如果同时满足这两个属性，就可以在不影响既有代码的前提下扩展功能。

③ 为什么 灵活应对代码的修改

不论什么软件，只要它还在生命周期内，就一定会发生变化。而且软件的寿命远比我们想象的要长。因此，我们设计出的软件要既能适应变化，又能保持长期的稳定。

这就要求代码能够灵活应对变化，对扩展开放，对修改关闭。如果能够满足上述要求，就算需求发生变化，我们只要给代码添加新的行为，就能毫无风险地完成对软件的修改。

如果设计得太过死板，那么一个小小的修改也会影响到所有与其存在依赖关系的部分。死板的设计是非常脆弱的。

46

④ 怎么做 给代码设置接口

我们要给代码添加接口。

在设计具有某项功能的模块时，如果让模块的使用者客户端直接调用模块的提供者服务器，我们就可以说这个设计是死板的，因为在这种情况下如果想使用其他服务器，还需修改客户端。

客户端 —使用→ 服务器

因此，我们要在客户端与服务器之间为模块的使用者设置“客户端接口”。这个客户端接口由服务器实现。

客户端 —使用→ 客户端接口 ←实现— 服务器

这样一来，在添加拥有新功能的服务器时，只要该服务器上有客户端接口，客户端就可以直接调用新服务器，我们就不用再修改代码了。也就是说，我们可以在不修改当前代码的前提下添加新功能。

⑤ 扩展一 OCP 的适用范围

OCP 的适用范围是有限的，我们不能要求所有代码都遵循 OCP。之所以这么说，是因为在没有发生变更的情况下，OCP 只会让代码变得复杂、冗长。

要想避免滥用 OCP，我们就需要做到不过分预估变更内容。然而，人是不可能完全正确地对变更内容进行预估的。

47

对此，我们可以先不管 OCP，等内容实际发生变更后再进行处理。也就是说，先豁出去挨第一发子弹（变更），然后进行修改，从而避免以后在同一个位置挨子弹。

使用 OCP 的关键在于预估，不过这里预估的不是变更内容，而是可能会发生变化的部分。以图形处理软件为例，图形的种类就是一个容易发生变化的部分，这一点我们不难想到。我们虽然不知道下次要添加的是五边形还是圆形，但可以预估出图形的种类会发生变化。找出可能发生变化的部分，将该部分藏到接口后面，这种手法叫作流动元素的胶囊化。

扩展二 OCP 的实现与设计

面向对象的多态性是实现 OCP 的代表技术。

不过，OCP 并不专属于面向对象。比如编译器在连接时，如果能切换要连接的库，则可以在非面向对象的语言中实现 OCP。OCP 的适用范围不受语言的限制。

另外，在“设计模式”这类设计手法中，有很多设计模式可以用来实现 OCP，具有代表性的有 Strategy、Observer、Template Method 和 Decorator。

⑥ 关联信息 受保护变化

GRASP（General Responsibility Assignment Software Pattern，通用职责分配软件模式）是一种职责驱动设计方法。该方法中介绍了一个名为“受保护变化”（protected variations）的设计模式。这个设计模式具体来说就是识别出安定部分与不安定部分的交界点，然后用安定的接口将交界点包裹起来。

也就是说，不安定部分中的不安定因素导致变更多发，受保护变化则用“接口”这一防护墙将变更带来的影响抵御在外。这样一来，接口就像防止火灾蔓延的防火墙和防止水漫四方的防水堤一样，保护软件不受变更的影响。

48

⑦ 出处

[1] 罗伯特 · C. 马丁. 敏捷软件开发：原则、模式与实践 [M]. 邓辉，译. 北京：清华大学出版社，2003.

⑧ 相关图书

[1] Bertrand Meyer. Object-Oriented Software Construction[M].Upper Saddle River:Prentice Hall, 1997.

[2] 埃里克 · 弗里曼，伊丽莎白 · 弗里曼，凯西 · 赛拉，伯特 · 贝茨.Head First 设计模式 [M].OreilyTaiwan 公司，译. 北京：中国电力出版社，2007.

[3] 克雷 · 拉蒙. UML 和模式应用 [M]. 李洋，郑龚，译. 北京：机械工业出版社，2006.

[4] 布莱恩 · 柯尼汉，罗勃 · 派克. 程序设计实践 [M]. 北京：人民邮电出版社，2016.

[5] 艾伦 · 沙洛维，詹姆斯 · R. 特罗特. 设计模式解析 [M]. 北京：人民邮电出版社，2010.

49



0.3 编程术语在本书中的用法

下面来介绍一下各编程术语在本书中的用法。如果对术语的理解存在偏差，就无法正确理解原则的含义，希望大家能多加注意。

① 软件

我们把编程的产物统称为“软件”。“应用”“系统”和“工具”等词语通常也指编程的产物，但在本书中我们一律称之为软件。

虽然在某些语境下使用“应用”这一叫法更为准确，但是鉴于书中介绍的原则既可运用于应用软件（应用），又可运用于基本软件（操作系统），所以本书中将编程的产物统称为软件。

② 代码

我们把编程中编写的东西统称为“代码”。“源代码”“源”“程序”和“源程序”等词语虽然也能表达相同的意思，但在本书中我们一律称之为代码。

需要特别注意的是“程序”一词。程序虽然经常用来指代编程中编写的东西，但有时它也可以用来指代编程的产物（可执行的软件），这就容易让人混淆，因此我们统一使用“代码”来表示编程中编写的东西。

③ 编程

我们把编写代码这一行为统称为“编程”。

虽然“编码”一词也能表达相同的意思，但它在部分人的印象中仅表示将自然语言转化为编程语言，因此本书中我们使用“编程”来表示编写代码这一行为。

另外，“实现”也有编程的意思。但在本书中，“实现”一词缩小了指代范围，仅表示接口或模块内的代码。

④ 程序员

我们把编程的人统称为“程序员”。虽然根据上下文也可以把编程的人称为“开发者”“开发人员”“工程师”“软件工程师”“编码员”等，但在本书中，我们一律称之为程序员。

⑤ 函数

我们把可供调用的代码块统称为“函数”。虽然这类代码块还能细分为例行程序、功能、过程、信息、方法和成员函数等，但在本书中，我们一律称之为函数。

当然，这些词语指代的东西并不完全相同。比如功能有返回值，过程就没有返回值。方法和成员函数则是面向对象语言中使用的术语。

不过考虑到简洁性，本书中我们统一将其称为函数。

⑥ 模块

我们把整合了函数、变量的责任单元统称为“模块”。虽然“类”“(集合了函数、变量的)文件”“组件”“部件”和“库”等词语也可以表示“关联性较强的代码的整合体”，但在本书中，我们一律称之为模块。

⑦ 软件架构

我们把软件的整体结构统称为“软件架构”。软件架构主要指比模块级别更高的软件整体的构造设计。另外，该术语也包含了“设计思想”的意思。



0.4 注意事项

本书对原则的说明具有三个特征，希望大家在理解这些特征的基础上对原则加以运用。

① 重复只因重要

在讲解的过程中，相同的内容会重复出现。越是看起来理所当然的内容就越会一遍一遍地出现。

这是因为许多原则虽然在观点和出发点方面有所不同，但关键部分是相通的，所以最终会得出同一个结论。

也就是说，越是重复出现的内容就越重要。正因为重要，它们才能够变换着形态，以不同的形式留存在各个原则之中。

这些“理所当然”的内容相信读者都知道，也都理解，但笔者还是希望大家能耐着性子读一读。在阅读的过程中不妨检查一下有哪些地方被自己忽略掉了。

② 自己思考实用的代码范例

本书不提供代码。虽然本书的目标读者是程序员，但笔者并没有通过具体代码来介绍编程的原则。虽然某些章节中有具体范例，但那些范例（包括代码）只用来帮助读者加强理解，不能用在实践中。

这么做是为了扩大原则的适用范围。一旦提供了具体的代码范例，读者的理解就会禁锢在这个范例里。另外，提供范例也会让人养成不主动思考的习惯。为防止此类情况发生，笔者会尽量抽象地介绍本书内容，然后把所有具体的运用方案留给读者去思考。

请不要误会，这并不是在否定具体编程技术的重要性。相反，具体

的编程技术更为重要。只不过先学习抽象的内容再学习具体技术效果更好而已。

希望读者能在了解原则的基础上深入学习具体的技术。

③ 若原则之间的主张相悖则取其平衡点

在本书介绍的原则中，原则之间的主张会存在相悖的情况。

对此，我们不能说某一方正确或者某一方错误，当然也不存在谁比谁更重要的情况。我们只能根据具体情况来看哪一个原则更加适用。

问题解决方案的有效程度由具体情况（人、目的、时间、来龙去脉、限制等）决定。此时应先着眼于大局，尽量收集信息，想出更多的解决方案。在此基础上恰当运用原则，从战略角度出发，选择当前应该重视的东西。

在这种情况下，我们追求的应该是“更优解”，而不是“最优解”。追求“更优解”能帮助我们迅速地找出符合当前情况的、平衡性良好的解决方案。




第 1 章　前提

~ 编程永恒的真理 ~

如果一个人不知道他要驶向哪个码头，
那么任何风都不会是顺风。

——塞涅卡






编程没有银弹

英 语 No Silver Bullet in programming

是什么 编程没有特效药

狼人是坊间传说的一种恐怖魔兽，它能将我们平日里司空见惯的东西瞬间变成恐怖的模样。能镇住狼人的方法只有一个，那就是将“银弹”射入它的体内。

在软件开发中，编程工作有时会突然陷入混乱。但可惜的是，没有“银弹”可供我们解决编程中的诸多问题。

编程中没有魔法般的万能解决方案，没有包治百病的特效药。

为什么 软件在本质上具有难度

作为编程产物的软件在本质上具有难度。

软件的本质是指软件不可欠缺的性质。一旦欠缺，软件就不再是软件了。在软件的本质中，有四种性质体现了其难度，这四种性质分别是复杂性、同步性、可变性和不可见性。

- **复杂性**

  软件是庞大且复杂的。几千万行的代码并不少见。

  软件各组成要素之间的依赖关系也随着软件规模的扩大呈非线性增加。软件比其他任何构造物都复杂。

- **同步性**

  软件必须与现实世界同步。

软件并不是单独存在的，它与硬件、网络、其他软件、人类的行动与习惯等现实世界的大量事物相关联。软件与现实世界相连，并在现实世界中被使用。

现实世界的复杂性决定了编程存在难度。

- **可变性**

软件必须时刻保持变化。

即便软件按照计划顺利交付，用户也会提出更多的要求，因为成品软件会使世界发生改变。软件对用户的认知造成影响，进而使用户产生新的需求。

无法驻足于安宁之地是软件的宿命。

- **不可见性**

软件是概念的集合体，而概念是肉眼看不见的东西。产品、进程、决策的过程等也都是不可见的。

虽然可以将软件抽象为单纯的画面，但其间信息会进行舍象，因此软件无法全部表现出来。

软件的不可见性提升了软件的难度。

本质，是定义上无法舍去的东西。既然软件在本质上具有难度，那么以软件为成果的编程必然也困难重重。编程所处环境较为复杂，问题涉及面也很广，因此不存在可以解决所有问题的特效药。

怎么做　研习历史，勇斗“复杂”

我们要学习软件开发的历史，脚踏实地地对软件进行改善。

很多人相信软件的世界中存在适用于一切情况的工具或技术。然而编程领域并不存在如“银弹”一般的万能技术或有效方法。不要被“特效药”的流言所迷惑，期待这种子虚乌有的解决方案。我们要脚踏实地，通过科学的途径对软件进行改善。

软件在本质上具有难度，其最大的原因就是软件具有复杂性。软件

开发的历史可以说是与复杂性相抗争的历史，其间诞生了许多对抗复杂性的策略。

我们要研习历史，学习各种方法和思路，努力削减复杂性给软件带来的影响。

扩展 软件非本质部分的改善

事物皆由本质部分和非本质部分构成。本质是指事物不可欠缺的性质，欠缺了本质，事物就不再是那个事物了。而非本质是指次要的、附属的性质，即使欠缺了这些性质，事物也还是那个事物。

软件开发现场需要用到的技术大多属于非本质的东西。构建（build）环境、编程语言、库和框架等都是软件的非本质部分。

软件非本质部分的活动对于推进软件本质部分的活动有着巨大的贡献，因为相对于本质部分，非本质部分更容易改善。比如，选择恰当的编程语言能够大幅提升生产效率。

在软件非本质部分的改善中，成效最大的当属自动化。测试、构建、环境搭建等的自动化大幅改善了工作效率和工作质量。我们要对软件的非本质部分进行自动化处理，尽量多留些时间给软件的本质部分。

出处

[1] 弗雷德里克・布鲁克斯. 人月神话 [M]. 汪颖，译. 北京：清华大学出版社，2002.

相关图书

[1] 大槻繁. ソフトウェア開発はなぜ難しいのか～「人月の神話」を超えて [M]. 東京：技術評論社，2009.

[2] 罗伯特・格拉斯. 软件工程的事实与谬误 [M]. 严亚军，龚波，译. 北京：中国电力出版社，2006.

[3] 史蒂夫・迈克康奈尔. 代码大全（第 2 版）[M]. 金戈，汤凌，陈硕，张菲，译. 北京：电子工业出版社，2006.



代码即设计文档

英 语 Code as design

是什么 代码才是设计书

在工程过程方面，软件和硬件经常被拿来比较。

硬件需要先设计，输出设计图，然后将设计图用作输入，进行物理上的“制造”。

将这一过程类比到软件中，就相当于上游工程负责人先通过设计创建出基本设计文档，然后将设计文档用作输入，由程序员编写代码完成“制造”。

然而这一类比存在错误。不管是上游的基本设计，还是详细设计、编程、测试和调试，这些都属于设计环节，输出的设计文档就是代码，而构建发布版本属于制造环节。也就是说，负责制造的不是程序员，而是编译器或构建系统。

从这个角度来说，编程属于设计行为。在软件开发过程中生成的所有文本，其中能真正称得上工程文档的，只有代码。

所以说，作为设计行为产物的代码才是设计文档。

为什么 改善对象是代码

产品的改善在设计环节进行。产品的性能无法在制造环节得到提升。制造环节和生产环节是按照设计制作产品的过程。

编程阶段会有许多内部代码和外部功能方面的改善，因此编程属于设计行为。

另外，基本设计、详细设计、编程、测试和调试是一个不可分割的整体，各项任务相互依存。拿基础设计来说，很多东西只有在开始编程之后才能搞清楚。位于上层的设计文档必须等到编程结束之后才能确定下来，这就证明了这几项任务是一体的。

设计输出的只有代码。CASE 工具和 UML（Unified Modeling Language，统一建模语言）虽然能够起到辅助作用，但设计最终必须使用编程语言来表现，然后经构建和测试来验证、优化。上游设计和下游设计只能使用编程语言作为共同的表现方式。

怎么做　优秀的程序员必不可少

下面来看一下为什么说编程是一种设计行为。弄清楚这一点后，我们就能看清编程的本质，确定开发的体制与环境了。

首先要明白，编程是一种具有创造性的行为。在有些人的眼中，编程只是把需求符号化了而已。然而，编程不是机械性的劳动，设计需要创造力和技艺。新人初来乍到，不可能一下子就上手，这种工作应由经验丰富的优秀程序员来完成。

另外，基本设计、详细设计、编程、测试和调试是设计中不可分割的任务。将这些任务分割开并不是明智之举。也就是说，全体程序员要共同参与设计，一起来编写代码。当然，我们也可以想办法将设计图自动变为软件，但这一方案并不现实。与其将来找人把不依赖于编程语言的设计图翻译成代码，倒不如让设计者自己编写代码来得省心。

更重要的是，既然代码属于设计的范畴，我们就应该尽早开始编写代码。不写代码的话就无法弄清很多事情，设计也只会无限拖延下去。

关联信息　罗塞塔石碑

代码是设计文档，但代码不是唯一的文档。除了代码，还有许多文档是必不可少的。

很多人认为敏捷开发是一种轻视文档的开发过程。但实际上，敏捷

开发只是不生成无用的文档，并没有主张“不生成文档”。

在具有持续性的编程活动中，最受重视的是一个叫作“罗塞塔石碑”的文档。这是一份写给未来维护负责人的简单手册，其中描述了用于理解软件开发环境和软件构架的信息。

在软件开发环境方面，文档描述了构建与测试过程的执行方法。构建中包含的测试能防止维护负责人在理解软件时落入陷阱。

在软件架构方面，文档中包含了一些能纵观全部代码的图表。

代码能很好地表达“怎么做”和“是什么”，却不能表达“为什么”，也就是设计理由。将设计理由描述在文档中，能为维护负责人提供判断材料。这种做法能在很多情况下起到作用。

出处

[1] 罗伯特・C. 马丁. 敏捷软件开发：原则、模式与实践 [M]. 邓辉，译. 北京：清华大学出版社，2003.

相关图书

[1] 凯佛林・亨尼. 程序员应该知道的 97 件事 [M]. 李军，译. 北京：电子工业出版社，2010.

[2] 埃里克・埃文斯. 领域驱动设计 [M]. 赵俐，盛海艳，刘霞，译. 北京：人民邮电出版社，2010.

[3] 肯特・贝克. 解析极限编程 [M]. 唐东铭，译. 北京：人民邮电出版社，2002.



1.3 代码必然被修改

英 语 Code will be changed

是什么 代码总是要修改的

代码不是写完就结束了，它在日后必然会被修改。没有写完就扔的一次性代码。

在编写代码的时候，我们应将“代码会被修改”这一点作为进行判断和选择时的优先考虑事项。

为什么 代码是无常的

软件在本质上具有复杂性，这就决定了它不可能是完美无缺的。软件在发布后必然会发生故障，这时我们就需要对故障进行修复。

另外，用户可能在软件发布后产生新的需求，因为有些问题只有等到用户实际使用软件之后才能被发现。任何软件都不可能在首次发布时就满足用户所有的需求。

除用户自身之外，用户所在商务环境的变化也会导致需求发生变化。软件必须迎合这种变化。如果执着于最初编写的程序，做出没有人用的软件，那么一切都是徒劳。

换个角度来看，开发新软件的编程过程其实也可以看作一种“修改”。在编程时，我们会给昨天编写的代码添加新代码，会为了让代码更易读而重构。从团队的层面来讲也是一样的，比如第一次迭代中的代码会在第二次迭代中被修改。不管在哪个阶段，代码都会因各种各样的理由而被修改。

怎么做　编写经得起修改的代码

编程中的任何一个判断都要以代码会被修改为前提。也就是说，编写的代码要经得起修改。

因此，提高代码的可读性就显得尤为重要了。代码这种东西，读远比写要费时间。如果以代码会被修改为前提，那么不管写代码需要耗费多少时间，只要读代码的时间能够缩短，我们就能把消耗在写代码上的时间赚回来。

出处

[1] 安德鲁・亨特，大卫・托马斯．程序员修炼之道 [M]. 周爱民，译．北京：电子工业出版社，2011.

相关图书

[1] 大槻繁．ソフトウェア開発はなぜ難しいのか～「人月の神話」を超えて [M]. 東京：技術評論社，2009.

[2] 肯特・贝克．解析极限编程 [M]. 唐东铭，译．北京：人民邮电出版社，2002.

[3] 凯佛林・亨尼．程序员应该知道的 97 件事 [M]. 李军，译．北京：电子工业出版社，2010.




第 2 章　准则

~ 编程的指导方针 ~

在我们尚未给予自己支配本身的力量时，
解放我们精神的所有行为，都是百害而无一益的。

——歌德






2.1 KISS原则

英 语 Keep It Simple, Stupid

或 Keep It Short and Simple

中 文 保持代码简洁

是什么 保持代码简洁

编写代码时，要优先保证代码的简洁性。

不管是从零开始编写代码，还是修复故障或扩展功能，都要注意保持代码简洁。

为什么 代码会越来越没有秩序

随意修改代码会使代码变得越来越复杂，越来越没有秩序。

复杂的代码可读性较差且难以修改。强行修改不仅会降低代码的质量，还会浪费时间。这样一来，我们就无法保证能在合适的时间发布修正版或者对软件进行更新。如果我们没有重视这个问题，依旧强行修改代码，代码就会变得没有人能看懂，最终腐化为无用之物。

而一份简洁的代码，其各个组成要素也是简洁的，各要素承担的职责也都降到了最小，各要素之间的关系也比较简单。因此，简洁的代码可读性高，容易理解，便于修改。各要素职责明确，使得测试也变得简单易行。程序员之间能更加轻松地通过代码进行交流，减少了在现实世界中多余的对话，节约了交流成本。这样，我们就能保证在不降低开发速度的情况下对软件进行长期维护。

代码必然会被修改，因此易于修改的特性对代码来说不可或缺。保持代码简洁可以使代码拥有易于修改的特性。

怎么做 不要画蛇添足

我们要将简洁视为编程的指南针。尽可能将多余的、过剩的要素从代码中剔除。

为此，在编程的过程中我们要经常问自己程序运行必不可少的要素有哪些。

听起来唯心，但这一点其实很重要，因为一旦放松警惕，代码就会变得复杂。比如下面这几种情况。

- **试图使用新学会的技术**

  学会一门新技术后，人们倾向于使用新技术写出一些无谓的代码。但是，代码并不是用来炫耀聪明才智的，它的作用是给用户提供价值。我们不能在代码上耍聪明。

  我们要多多斟酌代码的写法，努力保持代码简洁。

- **以备将来之需**

  有时人们觉得将来会用到某些功能，认为最好趁现在写下来，于是编写了过剩的代码。

  现在用不到的东西就不应该现在写，因为在大多数情况下，这些东西将来也用不到。

  我们应该只写当前需要的代码，保持代码简洁。

- **擅自增加需求**

  程序员有时会擅自增加需求，添加多余的代码。他们觉得，某个需求必要与否、正确与否，与其找用户确认，不如自己直接写出来。但是，需求是由用户决定的，程序员不可以擅自增加。

  一旦添加了不必要的代码，花费在维护上的时间就会像滚雪球一样增加。不写多余的代码是保证代码简洁的秘诀。

扩展　KISS 原则的适用范围

在软件开发中，KISS 原则不仅适用于代码，还适用于功能设计。

功能繁多的复杂软件不会得到用户的青睐，功能简洁、接口简洁的软件才会受欢迎。

另外，KISS 原则原本就适用于一切工程。只不过产物的复杂化倾向在软件开发中表现得尤为突出。造成这种情况的原因之一是人们认为软件开发能相对灵活地应对需求。

关联信息一　less is more

less is more 的意思是“少就是多”。这句话出自建筑领域，指通过减少装饰等表层要素和空间结构等内部要素，使建筑物能够经受住各种外在因素的影响，产生更加丰富的空间。

这个观点同样适用于软件。不写多余的代码，使代码保持简洁，防患于未然。另外，通过这种方式写出来的代码干净整洁，颇具美感。

我们要时常审视自己写的代码，问问自己这些代码是否多余。

关联信息二　奥卡姆剃刀

奥卡姆剃刀原理主张不应假设非必要的前提来说明某个事物。换句话说，就是当一个事物存在多种解释时，最简单的那个解释是正确的。该原理能帮助我们在理解事物时减轻思考方面的负担。

这个原理对于编程同样有效。在可以实现相同功能的情况下，不含多余内容的代码更能减轻阅读者的负担，可读性更高，也更易于改善。

出处

[1] 埃瑞克· S. 理曼德.UNIX 编程艺术 [M]. 姜宏，何源，蔡晓骏，译. 北京：电子工业出版社，2011.

相关图书

[1] 安德鲁·亨特，大卫·托马斯. 程序员修炼之道 [M]. 周爱民，译. 北京：电子工业出版社，2011.

[2] 理查德·蒙森·哈斐尔. 软件架构师应该知道的 97 件事 [M]. 徐定翔，章显洲，译. 北京：电子工业出版社，2010.

[3] 凯佛林·亨尼. 程序员应该知道的 97 件事 [M]. 李军，译. 北京：电子工业出版社，2010.

[4] 尼尔·福特. 卓有成效的程序员 [M]. 熊节，译. 北京：机械工业出版社，2009.

[5] 肯特·贝克. 解析极限编程 [M]. 唐东铭，译. 北京：人民邮电出版社，2002.



DRY

英 语 Don't Repeat Yourself

中 文 不要重复

是什么 严禁复制粘贴代码

不可以重复写相同的代码。

将整个逻辑随便复制粘贴到其他地方去用是造成代码重复的主要原因。这样一来，同一个逻辑将出现在多个地方。

```
func A() {
  Proc 1
  Proc 2
  Proc 3
```

```
func B() {
  Proc 4
  Proc 2
  Proc 5
```

条件相同的控制语句的代码块有时会重复出现在代码各处，这也是复制粘贴所致。当条件分支相同但分支后的处理不同时，如果先复制粘贴代码块再单独修改处理部分，就会出现这样的结果。

```
func C() {
  if(a) {
    xxx
  } else if(b) {
    xxx
  } else if(c) {
    xxx
}
```

```
func D() {
  if(a) {
    ooo
  } else if(b) {
    ooo
  } else if(c) {
    ooo
}
```

直接将常量写入代码也会造成代码重复。如果意义相同的常量在多

处使用，常量表达的信息就会重复出现多次。

```
func E() {
        ⋮
   if(值 < 16 ) {
   ...
   }
        ⋮
   if(值 < 16 ) {
   ...
   }
```

另外，用于对代码进行说明的注释虽然不属于纯粹意义上的代码重复，但它也是重复的一种。将代码翻译成母语的注释就属于这一类。

编写代码时要注意避免上述情况出现。一旦发现重复的代码，要立刻予以消除。

为什么　代码无法得到改善

代码一旦出现重复，故障修复、添加功能等，代码的改善措施就会变得难以实施。具体来说，我们会遇到以下困难。

● 代码的可读性下降

相同的代码出现多次，从量的角度来看是“代码量变大”，从质的角度来看是“复杂度变高”。显然，代码的可读性会下降。

无法准确理解代码就无法确立修改方针。

● 代码难以修改

当相同的代码出现在多处时，只有正确修改每一处代码，才能确保整体的一致性。稍有不慎，修改就会出现遗漏。

另外，即使代码完全相同，有时某些地方也用不着修改。在这种情况下，我们就需要阅读前后代码，判断这一处是否需要修改。

若当前重复的代码之间存在细微差别，我们就需要更加深入地阅读各个位置的代码。控制语句的条件内容或条件数量只要存在一点点差别，理解的难度就会进一步增大。弄不好代码会因无法解读而得不到改善。

- **没有测试**

  出现重复的代码大多是遗留代码，也就是说，这部分代码没有经过任何测试。

  在没有测试的状态下，就算我们拼尽全力去修改遗留代码，发生新故障的概率还是很大。

就算克服了上述所有困难，费尽九牛二虎之力完成修改，这些代码也会因为动了多个“大手术”而变得更加混乱。长此以往，当混乱蔓延至所有代码时，修改就会变成一个不可能完成的任务。

怎么做 将代码抽象化

我们可以通过对代码执行抽象化操作来消除重复。

对代码的逻辑执行抽象化操作，其实就是给整个处理命名，将其函数化、模块化。至于数据，则需要起个名字定义为常量。最后将重复的部分全部置换为抽象后的内容。

抽象化有以下几个优点。

- 减少了代码量，减轻了阅读负担
- 因为逻辑和数据有了名称，所以代码的可读性变高了
- 重复的代码集中到了一处，我们只对这一处进行修改即可。于是，代码的修改操作变得简单，代码的质量也得到了保证
- 抽象化的部分易于重复使用。在添加新功能的时候，重复使用代码可以更快、更好地完成编程

不过，执行抽象化操作需要我们跨越心理方面的障碍。比如将逻辑

转化为函数的操作就相当费时间，我们需要有足够的耐心。另外，由于我们修改的是原本可以运行的代码，所以修改后的代码存在不能正常运行的风险。抽象化操作还有一个最明显的缺点，那就是太麻烦。

然而，避免重复这一点没有商量的余地。从长远看来，避免重复的利大于弊，这是历史总结出来的结论。所以，即便要花时间重构，即便要花时间消除代码不能正常运行的风险，即便操作起来有些麻烦，我们也要消除重复的代码。

扩展一 DRY 的适用范围

DRY 不仅适用于代码，还适用于有关软件开发的一切活动。

比如软件开发作业中存在大量的重复劳动。DRY 也适用于这些重复作业。

具体来说就是将重复作业自动化。这样就可以消除手动作业的负担，同时保证操作的正确性，排除人为的离散因素。

最具代表性的自动化作业是对软件进行测试、构建和发布的持续集成。持续集成就是由设置了特定任务的集成服务器准确且频繁地自动执行软件的构建、测试和部署等作业。除了能消除重复的手动作业，持续集成还有保证构建质量、使构建不再依赖于特定人物、帮助我们尽早发现问题等优点。

扩展二 DRY 与编程技术

与其说编程技术大多具备实现 DRY 的功能，不如说大部分编程技术以消除重复作为目标之一。比如结构化编程和面向对象编程中就包含了消除重复的技术。

同样，消除重复也是大部分设计手法的目标之一。设计模式就是具有代表性的一种设计手法，它提供了代码结构模式以达到重复使用代码的目的。从另一个方面来看，设计模式也可以说是一种防止重复思考（重复思考同一问题的解决方案）的手法。

一般来讲，技术和方法的诞生都伴随着一定的目的。我们在学习技术时，不仅要学习具体做法，还要思考这项技术的目的是什么，这样才能事半功倍。

扩展三 不得已的重复

软件开发中难免会有不得已发生重复的情况。

比如阻抗失配。在填补编程与其所用服务之间不协调的地方时，填补信息难免会发生重复。

阻抗失配原本是电学领域的术语，指素材之间由于阻抗不同引起反射，使能量无法顺利传递。这里用来比喻结构不同的二者在连接时的难度。

在软件的世界中，失配容易发生在不同抽象化风格的交界处。最典型的例子就是面向对象编程的类和关系数据库的表。当关系数据库与使用关系数据库的编程语言同时存在时，为填补二者的失配，需要在关系数据库端的表定义、代码端的表映射配置文件和源文件三处存放相同的信息。这种重复的发生实属无奈。

不过，我们并非没有对策来解决上述问题。比如，我们可以将信息集中存放在一处，然后想办法让其他信息自动生成。这样就能对信息进行统一管理了。

关联信息一 WET

WET 是与 DRY 相对的概念，它是“Write Every Time”和“Write Everything Twice”的缩略语，表示重复同样的事情。有趣的是，DRY（干）和 WET（湿）不管作为英文单词还是缩略语，意思都是相反的。

WET 常用于讽刺那些没有实现 DRY 的代码。

关联信息二 OFOP

OFOP（One Fact in One Place）的意思是一个地方只有一个事实。它是数据库理论设计中表设计的一项重要原则。

DRY 要求代码不能出现重复，而 OFOP 要求数据库中不能存储重复的数据。OFOP 原则可以防止数据冗长和数据不一致（更新异常）的情况发生。

要想实现 OFOP，“标准化”这一设计手法必不可少。在编程中，消灭重复的方法有很多，但对数据库而言，“标准化”是独一无二的设计手法。

关联信息三 OAOO

OAOO（Once and Only Once）的意思是有且仅有一次。简单来说就是不可以出现重复，与 DRY 的意思和目的相同。

不过，OAOO 的适用范围要比 DRY 小，它只能用在编程语法上。OAOO 强调的是代码不能重复以及不能有多余的代码。

关联信息四 遗留代码

重复出现的代码大多没有测试程序。没有测试程序的代码称为遗留代码。

过去说起遗留代码，指的是那些以前写的不容易理解的、难以修改的代码。而如今，为了对测试程序能带来质量保证这一点加以重视，人们又将遗留代码重新定义为“没有测试程序的代码”。

这样一来，没有测试程序的代码就全部变成了垃圾代码。就算代码整齐，结构严谨，只要没有测试程序，那它就是遗留代码。

这个定义或许给人一种过于严格的感觉。代码整齐确实是好事，但这并不意味着整齐就万事大吉了。之所以这么说，是因为在没有测试程序的情况下修改代码非常危险。如果有测试程序，我们就可以一边检验

代码一边快速修改代码。改善代码时也一样。如果没有测试程序，就无法得知代码到底是变好了还是变坏了。

因此，当遇到遗留代码时，我们首先要为其编写测试程序。就算代码不够优雅也没有关系。总之，不论用什么方法，一定要先测试再修改。为了保证代码质量，这个步骤必不可少。

出处

[1] 安德鲁・亨特，大卫・托马斯. 程序员修炼之道 [M]. 周爱民，译. 北京：电子工业出版社，2011.

相关图书

[1] 迈克尔・ C. 费瑟. 修改代码的艺术 [M]. 侯伯薇，译. 北京：机械工业出版社，2014.

[2] 保罗・ M. 杜瓦尔，等. 持续集成 [M]. 王海鹏，贾立群，译. 北京：机械工业出版社，2008.

[3] 凯佛林・亨尼. 程序员应该知道的 97 件事 [M]. 李军，译. 北京：电子工业出版社，2010.

[4] 縣俊貴. 良いコードを書く技術—読みやすく保守しやすいプログラミング作法 [M]. 東京：技術評論社，2011.

[5] 大槻繁. ソフトウェア開発はなぜ難しいのか～「人月の神話」を超えて [M]. 東京：技術評論社，2009.

[6] 細谷功. 具体と抽象—世界が変わって見える知性のしくみ [M]. 東京：dZERO, 2014.



YAGNI

英 语 You Aren't Going to Need It

中 文 你不会需要它

是什么 只写所需最低限度的代码

不能以“可能会用到”为动机编写代码。我们要在需要的时候写需要的代码。

人们不可能预测出软件的变化，写出超前的代码。我们一定要明确这一点，坚持只写当前需要的代码。

为什么 代码无法预测

即使事先写好了一大堆代码以应对各种情况，这些代码大多也派不上用场。

编程针对的是特定需求，所以再怎么追求通用性，总有无法满足的情况。考虑到代码的扩展性，有时人们会把自己认为有用的东西设计进去。可惜这些预想大多不会成真。不能成真，就意味着浪费了时间。

况且将扩展性纳入考虑的范围会进一步使代码变得复杂。无用的代码混在其中，使得代码的可读性变低，难以维护。

当然，如果这些为扩展性服务的代码将来能派上用场，那么它们还是有价值的。然而现实不可能如此理想。相反，随着时间的流逝，人们会忘记这些没用的代码为了什么而存在。这样一来，当初费时费力想出的设计就没有了用武之地，成了碍事的垃圾。

怎么做　只写当前需要的代码

比起通用性，我们更应该重视单纯性。

先把通用性带来的可重复使用的特性和扩展性放到一边，将“能用”这一点放到第一位。

当在多个设计方案中进行选择时，我们重点要看的是设计方案的单纯性，而不是通用性。不要选择标榜通用性的复杂方案，要选择以具体需求为基础的简单方案。

很多时候，简单方案的通用性更强。

即使需求增加，功能需要扩展，简单的代码也比通用的复杂代码更容易修改。

扩展　YAGNI 的适用范围

除代码之外，YAGNI 还适用于软件的功能。

丰富的功能看上去很吸引人，但仅凭预想创建出来的“没有必要的”功能不但没有人使用，还会令软件整体的使用方法复杂化。

关联信息　DTSTTCPW

DTSTTCPW 是“Do The Simplest Thing That Could Possibly Work”的缩写，意为在可行方案中选取最简单的方案执行。其含义和目的都与 YAGNI 相同。

在编程中，经验告诉我们“今天选择简单的方案执行，等明天需要修改代码的时候，再花费时间和精力进行修改”要比“今天选择复杂的方案执行，结果没派上用场”更能获得好的结果。多条原则都表达了这样的观点，足以证明其正确性。

不过，选择简单的一方并没有想象中那么容易。人们很难不去思考明天、下周，甚至下个月要写的代码。

但是，想得太远只会增加代码的复杂度，提高修改的成本。所以每次编程时都要想想这些原则，让自己回到原点。

出处

[1] 尼尔・福特. 卓有成效的程序员 [M]. 熊节，译. 北京：机械工业出版社，2009.

相关图书

[1] 江渡浩一郎. パターン、Wiki、XP～時を越えた創造の原則 [M]. 東京：技術評論社，2009.

[2] Ron Jeffries, Ann Anderson, Chet Hendrickson[M]. Boston: Addison-Wesley, 2000.



PIE

英 语 Program Intently and Expressively

中 文 编程要表达出意图

是什么 表达出代码的意图

在写代码时，明确表达意图十分重要，这与写诗、写随笔、写博客和写信是一个道理。

这是因为代码是写给人看的，而不是写给编译器看的。

因此，在写代码时要在表达上多花心思，将软件运行方式直观地传达给阅读代码的人。

为什么 代码是唯一的线索

代码是我们正确、完整地了解软件运行方式的唯一线索。

软件开发过程中虽然会创建诸多文档，但这些文档并不能帮助我们正确认识软件是如何运行的。

需求定义文档只描述了需要什么东西。

基本设计文档只描述了用什么样的软件来实现需求。

详细设计文档只描述了成品软件是什么样的结构。详细设计文档虽然与代码最接近，但代码是动态变更的，而详细设计文档往往做不到同步，更何况并非每个项目都有详细设计文档。所以说详细设计文档并不是百分之百存在且百分之百有用的。

到头来，我们只能通过阅读代码来掌握软件的运行情况。因此，编写可读性高的代码，用代码表达意图是唯一可取的方法。

怎么做　把提高代码可读性作为第一要务

在编写代码时，我们要重视的是代码的可读性，而不是代码的易写性。

读代码的次数远比写代码的次数多。代码只写一次，但此后会被人多次阅读。所以，写代码的时间远没有读代码的时间长。包括自己正在写的代码，自己读的时间可能都要比写的时间长。

因此，“读代码的效率”应优先于“写代码的效率”。只要代码的可读性能够提高，就算牺牲写代码的效率也是值得的。只要有人使用软件，代码就会一直存活下去，所以代码的寿命往往超乎我们的想象。可读性高的代码，其价值也会随着时间的推移不断累积。

“读代码的效率”同样优先于“执行代码的效率”，因为如果代码的可读性较高，提高代码的执行效率也会变得容易一些。

将代码的可读性放在第一位，就意味着不能为了炫耀才华而写一些让人难懂的代码。刚学会新技术时，人们很容易落入陷阱，明知学会的新技术用不上还想炫耀一下。不能让人读懂的代码不是好代码。只有能够向读者准确传达意图的代码，才是能够帮我们达到目的的好代码。

不管是自己写的代码还是别人写的代码，只要我们在阅读的过程中不能立刻理解，就应该马上对其进行修改，提高代码的可读性。久而久之，工作就能顺利开展下去了。

扩展一　避免打地鼠式的开发

打地鼠是一个很有名的游戏。玩家手持木槌，敲击从土中露出脑袋的地鼠。没人知道地鼠下次会从哪里钻出来，这正是该游戏的乐趣所在。

然而在软件开发中，这并不是一件好事。修改一个会在各处随机出现问题的软件毫无乐趣可言。

软件的项目经理会面临各种催促交付的压力，对他们来说，时间非常重要。因此，他们更倾向于牺牲代码的质量来提高实现功能的速度。

这就是打地鼠式的开发。编写可读性高、没有故障的高质量代码

（和测试程序）确实需要花费很多时间，从短期来看是一种损失。但这样做能够避免打地鼠式的开发，从长远来看，利大于弊。

扩展二　要写注释

理想的代码，是可读性高到没有注释也能读懂的代码。

然而，代码毕竟只能表达“做什么”和“怎么做”。要表达“为什么这样做”，还需要用到注释。

代码文本的作用是与读代码的人进行交流。为了使交流更加顺畅，我们还要把注释当成一种交流的手段。我们在写代码时要找到一个平衡点，一方面要让代码在不需要注释的情况下能被人理解，一方面用注释来弥补代码表达不出来的东西。

关联信息　文学编程

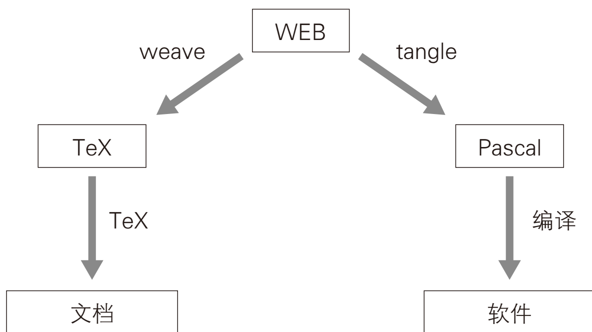
表达意图的编程中有一个终极形态，那就是“文学编程”（literate programming）。

文学编程是一种将代码本身当作文档的编程方法。在文学编程中，用于描述文档的语言与编程语言结合在了一起。

这里所说的文档是对程序的说明。在经过编译之后，这些文档就成了能够执行的软件。代码即文档，文档即代码。

也就是说，在文学编程中，代码与文档并不分开编写，二者会编写在一个文本中。然后，生成程序会区分代码部分和文档部分，将二者分别交给编译器和文档处理程序来处理。

实现这一功能的是文学编程的支持软件 WEB。这个工具能解析以文学编程方法写出的代码，生成（weave）以特定形式输出的文档的源文件（TeX），同时生成（tangle）可供一般编译器使用的源代码（Pascal）。



以文学编程的方法编写代码，可以让代码像故事一样写出来。因此，人们可以根据上下文轻松读懂代码。在文学编程中，代码的描述不受语言解释器的约束。在这里，代码可以像文学作品一样写出来。

文学编程具有以下优势。

- 由于不必另外编写文档，所以程序员可以一边编程一边描述代码的解释性内容及代码的正当性依据，这就使程序员可以从另一个角度来审视代码
- 代码离相关说明的位置很近，便于修改。修改代码后，相关说明也能及时得到更新
- 保证整个代码中只存在一个文档
- 代码中包含普通注释不会描述的信息（算法的说明、正当性依据、设计上的依据等）

软件的维护阶段最能体现文学编程的价值。由于可以直接参考丰富且高质量的说明信息，所以用文学编程方法写出的代码要比一般代码更容易维护。

然而从结果来看，文学编程的编程环境并没有得到普及。这是因为与其他编程语言相比，文学编程写代码的负担更重。

不过，文学编程中“代码要表达作品的意图，并且要有自我说明的能力”这一观点还是以 PIE 原则的形式保留了下来。

出处

[1] 文卡特・苏帕拉马尼亚姆，安迪・亨特. 高效程序员的 45 个习惯：敏捷开发修炼之道 [M]. 钱安川，郑柯，译. 北京：人民邮电出版社，2010.

相关图书

[1] 皮特・古德利弗. 编程匠艺——编写卓越的代码 [M]. 韩江，陈玉，译. 北京：电子工业出版社，2008.

[2] 有沢誠. クヌース先生のプログラム論 [M]. 東京：共立出版，1991.

[3] 芭比・戴维斯，项目经理应该知道的 97 件事 [M]. 张科，焦亚超，译. 北京：人民邮电出版社，2011.

[4] 达斯廷・博斯韦尔，特雷弗・富彻. 编写可读代码的艺术 [M]. 尹哲，郑秀雯，译. 北京：机械工业出版社，2012.



2.5 SLAP

英 语 Single Level of Abstraction Principle

中 文 单一抽象层次原则

是什么 统一代码的级别

在编写代码时，我们要将高级别的抽象化概念和低级别的抽象化概念分离。

在分离时不能只有高低两层。我们要根据功能的复杂程度对抽象化概念进行分离，然后统一各层的抽象级别。

也就是说，我们要根据抽象级别对函数进行分割，并且将同一函数中的代码统一为同一个抽象级别。

统一各抽象级别之后，代码便成了一本优秀的图书。优秀的图书结构紧凑，各章节的分割与排版都很严谨，通读下来能获得最流畅的阅读体验。当然，我们也可以将其视为参考资料，从任意位置开始阅读。

统一各抽象级别之后，代码就可以像图书一样供人阅读了。高级到中级的处理相当于图书的目录，级别最低的处理相当于图书的正文。

```
function 高级(){// 1级目录
         中级1();
         中级2();
}
function 中级1(){// 2级目录-1
         低级1();
         低级2();
}
function 低级1(){// 正文内容
         // 处理
}
```

```
function 低级2(){// 正文内容
         // 处理
}
function 中级2(){// 2级目录-2
         低级3();
}
function 低级3(){// 正文内容
         // 处理
}
```

为什么 使代码具有概括性和可读性

将代码分割成级别统一的函数，能使代码具有概括性和可读性。也就是说，函数一览起到了目录的作用，从而使代码拥有概括性。分割后的函数是小块的代码，这就提升了代码的可读性。

代码统一之后，抽象度相同的处理都在同一个地方。于是，代码变得更加顺畅，更容易让人理解。

相反，如果读到一半时代码的抽象度突然发生了改变，流畅感就会戛然而止。这就给阅读造成了障碍，并且会扰乱我们对前面代码的理解。

怎么做 将函数结构化

我们要将函数结构化。

将处理转换为意图清晰、由抽象化级别一致的多个步骤组成的函数。

将函数结构化之后，各函数的处理将以调用比自己低一个级别的函数为中心。这种由其他函数的调用组成的函数称为复合函数（composed method）。

复合函数要尽量小。如果想通过名称来表达意图，那么即使处理只有一行，也可以写成函数。

另外，不要在复合函数中调用不同抽象级别的函数。也就是说，一

个函数中不能既有“连接数据库”这种低级处理，又有“执行业务逻辑”这种高级处理。

扩展一 SLAP 的适用范围

除函数外，SLAP 还适用于模块等结构。

在优秀的软件设计中，概念被划分为多个级别，不同级别的概念会装入不同的容器中。

拿面向对象中的类设计来说，容器是抽象类和它的继承类。用抽象类存放级别较高的概念，用它的继承类存放级别较低的概念，由此便实现了 SLAP。

不过，此时要确保级别较低的概念只存在于继承类，级别较高的概念只存在于抽象类。

扩展二 实现 SLAP 的步骤

我们可以参考写文章时的步骤来实现 SLAP。

要写出一篇优秀的文章，关键在于将“编写内容”与“设计结构”这两件事分开来做。

这种做法同样可以套用到代码的编写上。也就是说，将编写具体处理的操作和统一抽象化级别的操作视为两个不同的工作，分别以不同的模式去实现。这样，我们操作起来会更加简便，代码的质量也会得到提升。

关联信息 代码与图书

优秀的图书能够为编写优质的代码提供参考。这里我们对图书的构成要素进行分析，看看各要素分别相当于代码的哪个部分，研究一下如何才能写出优质的代码。

● **序言**

图书的序言用于体现该书的主旨，从全局说明该书的立意。

序言部分在代码中相当于文件开头的注释。这部分注释负责说明代码描述的内容以及代码所属的项目。

● **目录**

图书的目录相当于代码的函数一览。

有人提倡在代码的开头列出代码中所有的函数。不过，考虑到集成开发环境以及编辑器的函数一览功能和跳转功能，这一做法其实没有什么必要。

● **部分**

图书的内容有时会分成多个部分。

在某些情况下，代码也可以分割成多个部分，比如一个文件中描述了多个模块，或者文件内的函数可以在逻辑上分成多个组等。这时就需要用注释来区分各个部分，明确表示出逻辑上的分段。

不过，在一个文件中描述过多的内容并不是明智的做法，因为这么做不仅会使内容变得混乱，让人难以理解，还会给文件内的移动操作带来麻烦。

因此，在代码量较多的情况下，分多个文件描述会比较好。

● **章**

图书的章是给一整块有头有尾的内容添加一个标题后形成的。

它相当于代码中的函数。我们要套用 SLAP 编写结构化函数。

● **段落**

图书中的段落相当于函数中的代码块。和图书中的段落一样，函数中的代码块也要用空行来区分。

各函数的逻辑由语句罗列而成。我们要将语句按照逻辑区分成块，通过空行来和后面的代码进行区分。空行在语法上没有意义，它仅仅用于提高代码的可读性。

- **正文**

  图书中的一句句话相当于代码中的一条条语句。与写文章一样，为了便于读者理解，语句要尽量简短。一条语句只描述一件事情。

- **注释、索引**

  一些图书也有注释和索引，但代码中不需要有相应的内容。集成开发环境以及编辑器的跳转和搜索功能可以满足这部分需求。

出处

[1] 尼尔·福特. 卓有成效的程序员 [M]. 熊节，译. 北京：机械工业出版社，2009.

相关图书

[1] Kent Beck. Smalltalk Best Practice Patterns[M]. Upper Saddle River: Prentice Hall, 1996.

[2] 肯特·贝克. 实现模式 [M]. 李剑，熊节，郭晓刚，译. 北京：人民邮电出版社，2009.

[3] 乔舒亚·科瑞夫斯盖. 重构与模式 [M]. 杨光，刘基诚，译. 北京：人民邮电出版社，2006.

[4] 皮特·古德利弗. 编程匠艺——编写卓越的代码 [M]. 韩江，陈玉，译. 北京：电子工业出版社，2008.



2.6 OCP

英 语 Open-Closed Principle

中 文 开闭原则

是什么 代码的修改不相互影响

我们要让代码同时满足对扩展开放、对修改关闭这两个属性。

对扩展开放表示代码的行为可以扩展。

对修改关闭表示当对代码的行为进行扩展时，其他代码完全不会受到影响。

代码如果同时满足这两个属性，就可以在不影响既有代码的前提下扩展功能。

为什么 灵活应对代码的修改

不论什么软件，只要它还在生命周期内，就一定会发生变化。而且软件的寿命远比我们想象的要长。因此，我们设计出的软件要既能适应变化，又能保持长期的稳定。

这就要求代码能够灵活应对变化，对扩展开放，对修改关闭。如果能够满足上述要求，就算需求发生变化，我们只要给代码添加新的行为，就能毫无风险地完成对软件的修改。

如果设计得太过死板，那么一个小小的修改也会影响到所有与其存在依赖关系的部分。死板的设计是非常脆弱的。

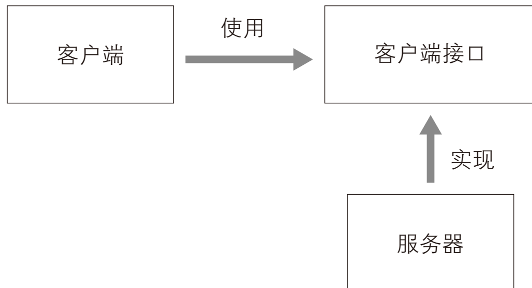
怎么做　给代码设置接口

我们要给代码添加接口。

在设计具有某项功能的模块时，如果让模块的使用者客户端直接调用模块的提供者服务器，我们就可以说这个设计是死板的，因为在这种情况下如果想使用其他服务器，还需修改客户端。



因此，我们要在客户端与服务器之间为模块的使用者设置“客户端接口”。这个客户端接口由服务器实现。



这样一来，在添加拥有新功能的服务器时，只要该服务器上有客户端接口，客户端就可以直接调用新服务器，我们就不用再修改代码了。也就是说，我们可以在不修改当前代码的前提下添加新功能。

扩展一　OCP 的适用范围

OCP 的适用范围是有限的，我们不能要求所有代码都遵循 OCP。之所以这么说，是因为在没有发生变更的情况下，OCP 只会让代码变得复杂、冗长。

要想避免滥用 OCP，我们就需要做到不过分预估变更内容。然而，人是不可能完全正确地对变更内容进行预估的。

对此，我们可以先不管 OCP，等内容实际发生变更后再进行处理。也就是说，先豁出去挨第一发子弹（变更），然后进行修改，从而避免以后在同一个位置挨子弹。

使用 OCP 的关键在于预估，不过这里预估的不是变更内容，而是可能会发生变化的部分。以图形处理软件为例，图形的种类就是一个容易发生变化的部分，这一点我们不难想到。我们虽然不知道下次要添加的是五边形还是圆形，但可以预估出图形的种类会发生变化。找出可能发生变化的部分，将该部分藏到接口后面，这种手法叫作“流动元素的胶囊化”。

扩展二　OCP 的实现与设计

面向对象的多态性是实现 OCP 的代表技术。

不过，OCP 并不专属于面向对象。比如编译器在链接时，如果能切换要链接的库，则可以在非面向对象语言中实现 OCP。OCP 的适用范围不受语言的限制。

另外，在设计模式这类设计手法中，有很多设计模式可以用来实现 OCP，具有代表性的有 Strategy、Observer、Template Method 和 Decorator。

关联信息　受保护变化

GRASP（General Responsibility Assignment Software Pattern，通用职责分配软件模式）是一种职责驱动设计方法。该方法中介绍了一个名为“受保护变化”（protected variations）的设计模式。这个设计模式具体来说就是识别出稳定部分与不稳定部分的交界点，然后用稳定的接口将交界点包裹起来。

也就是说，不稳定部分中的不稳定因素导致变更多发，受保护变化则用接口这一防护墙将变更带来的影响抵御在外。这样一来，接口就像

防止火灾蔓延的防火墙和防止水漫四方的防水堤一样，保护软件不受变更的影响。

出处

[1] 罗伯特・ C. 马丁 . 敏捷软件开发：原则、模式与实践 [M]. 邓辉，译 . 北京：清华大学出版社，2003.

相关图书

[1] Bertrand Meyer. Object-Oriented Software Construction[M]. Upper Saddle River: Prentice Hall, 1997.

[2] 埃里克・弗里曼，伊丽莎白・弗里曼，凯西・赛拉，伯特・贝茨 .Head First 设计模式 [M]. OreilyTaiwan 公司，译 . 北京：中国电力出版社，2007.

[3] 克雷・拉蒙 . UML 和模式应用 [M]. 李洋，郑龚，译 . 北京：机械工业出版社，2006.

[4] 布莱恩・柯尼汉，罗勃・派克 . 程序设计实践 [M]. 北京：人民邮电出版社，2016.

[5] 艾伦・沙洛维，詹姆斯・ R. 特罗特 . 设计模式解析 [M]. 北京：人民邮电出版社，2010.



2.7 名字很重要

英 语 Naming is important

是什么 命名是代码最重要的课题

在编程中，我们要将命名视为最重要的课题，谨慎对待。
“命名”这一行为和它的产物“名字”都具有非常重要的价值。

- **命名**

  取了一个合适的名字意味着元素被正确理解并被正确地设计了出来。相反，如果取的名字不合适，就证明程序员还没有充分理解该元素的作用。

  取了一个合适的名字就表示设计已经完成了一大半。

- **名字本身**

  程序员之间通过代码进行交流时，名字传递的信息是最多的。

  写代码的人和读代码的人很少能站在一起实时进行对话。程序员之间大多通过代码进行交流，一旦名字不够贴切，代码上的沟通就会出现障碍。

  为了让这种非实时对话能够顺畅进行，我们必须最大限度地在名字上下功夫。

为什么 名字是面向代码阅读者的“用户界面”

名字是面向代码阅读者的“用户界面”。各元素都有适当名字的代

码能有效传达其意图，让人充分理解某个东西是怎样做出来的。

以给函数命名为例，名字恰当易懂的函数能通过名字表达其职责，这有助于隐藏函数的内部处理。从结果来看，有以下好处。

- 在阅读代码时，只看函数名就能大致掌握其内部处理，因此可以跳着阅读内部代码
- 在编写代码时，函数名能帮助我们理解该函数的目的及用法，使函数调用变得简单。对编写完的代码来说，恰当的名字起到了说明作用，使代码的可读性大大提高

相反，名字晦涩难懂的函数会给代码方面的作业增加很大的负担。具体来说有以下几点。

- 在阅读代码时，如果无法通过函数名了解处理内容，就必须解析函数的内部代码。也就是说，读代码的人要被迫深入阅读代码。如果函数嵌套层数不是很多倒也还好，但随着层数的增加，程序员的负担会逐渐加重
- 在编写代码时，如果无法通过函数名进行各种判断，就需要对函数内部进行解析。即便大费周章完成了代码的编写，代码也会因所调用的函数的名字不当而看上去支离破碎，变得难以读懂

在读写代码时，程序员的大脑处于超负荷运转状态。程序员在面对代码时，如果大脑被“代码是怎么运行的”这一问题占满，就无法思考其他问题，导致工作停滞不前。

程序员并不是因为想读代码才去读的。在充分理解代码之后对其进行修改或添加功能才是他们真正的目的。在阅读代码时，混乱的名字会占用所有的脑部资源，妨碍原本应该进行的作业，使程序员无法着手处理问题。

琢磨名字需要花费很多精力。要想取一个恰当的名字，我们得有强大的思考能力，还要舍得花时间。相反，随便取一个名字不仅不会影响函数的运行，还能节约时间。

但是，不恰当的名字会让代码“负债”。此后，只要代码还在使用，读代码的人、用代码的人都将受到负面影响。

怎么做 写代码前先决定名字

编程要先从名字入手。先给代码中会出现的各个元素取一个能准确表达意图的名字。

在编写代码的过程中也要时常站在代码使用者和阅读者的角度命名。具体来说，我们要注意以下几点。

- 名字中要尽量多包含信息。将名字视为简短的注释有助于将必要的信息添加进去。可以多准备几个名字，从中选择最合适的一个。这样能进一步提高名字的质量
- 名字不能有歧义。命名之后，多问自己几遍这个名字是否存在歧义。要想减少歧义，就需要了解并遵循各编程语言的习惯
- 名字说明的是效果和目的，而不是手段。如果所取名字符合这一条件，读代码和使用代码的人就不用在理解代码内容上花费太多时间
- 可以通过先写测试程序后写处理的方式检查一下自己取的名字是否合适。这样能帮助我们从使用者的角度来审视名字合适与否
- 名字要能念出来。除了便于在现实对话中使用之外，能念出来的名字还可以减轻阅读代码时大脑的负担
- 名字要能搜索出来。名字如果是一个字母或是一个数字，搜索时就会在代码中产生无数个结果。这会给代码解析带来额外的负担

扩展 避免心理映射

将外界信息转换为记忆中原有形象的过程称为心理映射。

读代码的人看到某元素名字后需要先在心里把它转换成自己知道的东西，这个过程就是心理映射，我们应极力避免这种情况发生。对集中

注意力追踪代码流向的程序员而言，心理映射会给大脑带来极大的负担，分散注意力。

心理映射常发生在使用了不合乎规定或问题领域之外的术语等情况下。代码中若使用了独创的名字，读代码的人就必须先进行心理映射才能理解元素表达的意思。因此，我们要尽可能地用标准术语来命名元素。相比独创的名字，标准术语更能获得大家的共识，从而减轻读写双方的负担。

另外，在使用仅有一个字母的变量名（循环计数器除外）时也会发生心理映射。一个字母的变量名只是一个占位符，读代码的人需要在心里将其转换成相应的术语才能理解。

程序员的能力是用来为用户创造价值的。在命名方面炫耀知识是多余之举。我们应该使用大家能理解的名字，写容易理解的代码。

关联信息 环回检测

有一种叫作“名字可逆性”的命名思路。该思路主张名字必须能还原其所指内容的说明文本。

要想满足这一条件，就需要进行环回检测。先通过内容的说明文本来想名字，再通过名字倒推出说明文本。按照说明文本、名字、说明文本的顺序绕一圈回来（环回）后，如果说明文本一致，那这个名字就是好名字，如果不一致就需要我们注意了。

这里来举一个例子。我们先通过说明文本来想名字。说明文本的内容是“一种用语音来操作软件的功能”。

- **语音识别功能（×）**

  这个名字没有表达出识别语音后通过语音进行操作的部分。因此人们容易把它理解成一种通过语音输入数据的功能。

- **语音操作功能（△）**

  这个名字没有表达出语音和操作之间的关系。人们容易把它理解成一种操作语音的功能。

由此可以看出，直接使用原说明文中的单词并不能达到理想的效果。

- **语音控制功能（△）**

这个名字听起来让人觉得该功能是一种对语音加以限制的功能。

- **语音命令功能（○）**

这个名字基本不存在歧义。但“命令”可能会被解释为“输出语音”，而不是“输入语音”。

出处

[1] 松本行弘．松本行弘的程序世界 [M]．柳德燕，李黎明，夏倩，张文旭，译．北京：人民邮电出版社，2011.

相关图书

[1] 凯佛林・亨尼．程序员应该知道的 97 件事 [M]．李军，译．北京：电子工业出版社，2010.

[2] 開米瑞浩．ネーミングの掟と極意 [M]．東京：翔泳社，2017.

[3] 达斯廷・博斯韦尔，特雷弗・富彻．编写可读代码的艺术 [M]．尹哲，郑秀雯，译．北京：机械工业出版社，2012.

[4] 皮特・古德利弗．编程匠艺——编写卓越的代码 [M]．韩江，陈玉，译．北京：电子工业出版社，2008.

[5] 罗伯特・ C. 马丁．代码整洁之道 [M]．韩磊，译．北京：人民邮电出版社，2010.

第 3 章　思想

~ 编程的意识形态 ~

少数拥有特殊嗅觉的人能够发现全新事物，
但他们并非有独创性的人。
真正有独创性的人拥有一双慧眼，
他们能将众人眼中的旧事物、人尽皆知的事物，
大多数人觉得不值一提、熟视无睹的事物，
视作崭新的事物。

——尼采



3.1 编程理论

英 语 A theory of programming

是什么 指导编程的思想

在编程的过程中，人们最重视的莫过于编写出高质量的代码。

高质量的代码是指拥有多种扩展方法、不存在多余要素、可读性高、易于理解的代码。

编程中有一套理论专门用来指导人们实现这种高质量的代码。该理论由以下三个思想作为支撑。

- 交流
- 简洁
- 灵活性

在追求高质量代码的过程中，这些思想左右着我们的每一个决定。

为什么 将编程理论展示的思想作为技术的选择基准

编程在不同的问题领域有不同的技术和模式。

虽然不能否认理解和掌握技术的重要性，但单纯的学习只能帮助我们了解技术的表面，并不能帮助我们真正学会使用这项技术。

编程中解决问题的方式是就事论事。由于每次出现的问题都不相同，所以如果不先找出问题所在，我们就无法选择合适的技术来使用。要想熟练掌握一项技术，就得明白为什么要使用这项技术、这项技术有

什么价值，以及我们应该在什么时候使用它。

此时就需要用到编程理论了。编程理论所展示的思想就是使用各项技术的理由。

怎么做 通过六个原则将编程理论展示的思想应用于代码

我们要把编程理论展示的思想用作判断基准。这样做可以更加准确地判断技术和方法适用与否。在把这些思想应用于代码的过程中，我们可能会发现一些新的方法。

不过，把思想直接应用于实际编程未免有些抽象，所以我们需要一个“桥梁”来连接思想与编程。这里有六个原则可以充当二者之间的桥梁。

- 效应局部化
- 重复最少化
- 逻辑与数据的一体化
- 对称性
- 声明式表达
- 变动率

关于这三个思想和六个原则，笔者会在后面详细进行讲解。

关联信息一 视点

视点是我们使用某项技术时应该考虑的内容。视点是一种看待事物的方式，它能让我们的思想紧贴当前正在处理的问题。

以下几点内容都属于视点。

- 解决方案应满足的需求（例如需要进程间通信等）
- 课题中包含的限制（例如进程间通信必须符合协议标准等）
- 解决方案需要具备的特性（例如便于添加功能、添加的功能不影响现有功能等）

根据这些视点，我们能从各个角度探讨课题，能更深入地把握课题。

不过，视点与视点之间可能是相辅相成的，也可能是互相排斥的。我们要做的是掌握各视点之间的平衡。掌握了各视点之间平衡的方案就是该课题的优秀解决方案。

我们在选择代码的实现方法时，要将编程理论作为视点来使用。也就是说，要以思想为“动机”，以原则为“桥梁”来选择代码的实现方法。

关联信息二 现有的工具是如何形成的

程序员需要使用信息处理技术这一工具来解决各种各样的问题。这时，了解工具的形成过程就成了程序员熟练使用工具必不可少的一个条件。

有这样一句格言——形态遵从于功能。这句话是说工具的最终形态由目的决定。

技术也是一种工具。在解决问题的过程中，技术经过反复打磨才有了今天的模样。如果掌握了技术的使用步骤而不了解它的演变历史，那么就无法使用该技术真正解决问题。因为在这种情况下，所选技术与要实现的目标并不匹配。

另外，在不清楚技术为何而用的情况下，对于知识，我们也只会左耳进右耳出。如果不理解使用技术的目的，解决问题时就不会一帆风顺。

因此，在学习技术的同时，我们也要了解它的工作原理、演化过程和设计背景。只有这样，我们才更容易达成目的。

一名好的程序员，不论面对何种语言、工具、技术和问题，都会花时间充分对其进行理解，然后开展工作。在编程领域，“搞不懂原理也没关系，代码能运行就好”“搞不懂原理也没关系，故障排除了就行”的思想必然会带来质量方面的问题。所以在编写代码之前，一定要尽全力去理解眼前的语言、工具、技术和问题。

出处

[1] 肯特・贝克. 实现模式 [M]. 李剑, 熊节, 郭晓刚, 译. 北京: 人民邮电出版社, 2009.

相关图书

[1] 罗伯特・ C. 马丁. 代码整洁之道 [M]. 韩磊, 译. 北京: 人民邮电出版社, 2010.

[2] 有沢誠. クヌース先生のプログラム論 [M]. 東京: 共立出版, 1991.

[3] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal. Pattern-oriented software architecture[M]. New York: Wiley, 1996.

[4] James O.Coplien, Neil B. Harrison. Organizational Patterns of Agile Software Development[M]. Upper Saddle River, New Jersey: Prentice Hall, 2004.

[5] アスキー書籍編集部.UNIX MAGAZINE Classic with DVD[M]. 東京: ASCII, 2007.

[6] 西尾泰和. 代码之髓: 编程语言核心概念 [M]. 曾一鸣, 译. 北京: 人民邮电出版社, 2014.



支撑编程理论的三大思想①

交流

英 语 Communication

是什么　代码是交流的场所

代码也是一种给人看的文档，而文档的本质在于交流。

在编程中，良好的交流意味着读代码的人能够理解、修改和使用代码。

为什么　顺利的开发源于顺利的交流

软件开发的大部分成本是在开发完成后产生的。这部分成本就是维护成本。

要想节约维护成本，就需要提高代码的可读性。这是因为程序员之间需要通过代码进行交流。

读代码的时间远多于写代码的时间。软件从开发到寿终正寝，期间有很多人一遍又一遍地读其中的代码。

此外，除了在维护阶段需要读代码，在开发阶段程序员也需要一边回顾前面的代码一边编写新代码。通过代码，程序员与片刻之前的自己实现了交流。

怎么做　从代码阅读者的角度出发

要想通过代码取得良好的交流效果，我们在写代码时就得站在代码阅读者的角度思考。

在刚开始编写代码时，我们的思维很容易停留在如何编写出能让计算机正常处理的代码上。此时我们不妨改变一下视角，思考他人是如何看待这段代码的。这样一来，我们便能从新的角度审视问题，寻找解决方案。

把思考的重点从计算机转移到代码阅读者的身上，就是所谓的换位思考。代码不仅是输入到编译器和解释器中的信息，也是写给人看的“文章”。我们要重视代码在交流方面的作用。



支撑编程理论的三大思想②

简洁

英　语　Simplicity

是什么　消除代码的复杂性

对代码而言，简洁就是消除了“多余的复杂性”后的状态。这里所说的“多余的复杂性”不是指反映了目标（代码要达成的目的）复杂程度的复杂性，而是指在修改代码的过程中遗留下来的痕迹所带来的复杂性。

为什么　代码的复杂性是罪魁祸首

“多余的复杂性”不具有任何价值。这类复杂性会阻碍代码正常运行，提高修改代码的难度，损害软件的价值。它会给代码埋下祸根。

消除“多余的复杂性”可以让代码变得简洁。这样一来，阅读、使用、修改代码的人就能快速理解代码。简洁的代码还能大幅降低故障发生的概率。

怎么做　分清代码中的“玉”与“石”

要想让代码保持简洁，我们就需要将代码中的“玉”与“石”明确区分开来。在设计时，将代码的本质部分（玉）放在显眼的位置，保证其他元素（石）不会混入其中。

简洁能给交流带来好的影响。消除多余的复杂性后，留下来的本质

部分会更加显眼，代码也变得更易于理解。这就有利于提高交流的效率。同样，重视交流，从他人的角度看问题，能够帮助我们判断出应该剔除哪些东西，从而提高简化代码的效率。

不过，简洁与交流偶尔也会发生冲突。过度简洁会使代码变得难以理解。在遇到这种情况时，我们就需要牺牲一部分简洁性，把交流放在优先的位置。



支撑编程理论的三大思想③

3.4 灵活性

英　语 Flexibility

是什么　代码易于修改

代码的灵活性是指修改代码的难易程度。

“灵活”是指在添加新代码的时候，已有代码能够不受影响、不产生冲突、不出现排斥，在保证自身不遭到破坏的前提下灵活地接纳新代码。

为什么　代码必然会被修改

代码早晚会被修改。因此，易于修改、具有灵活性成了代码的必备条件。

软件开发并不是在软件发布之后就结束了。为了持续向用户提供服务，发布后我们还要对软件进行维护，比如修复故障、调整功能、添加新功能等。

修改代码的难易程度关系到维护工作能否顺利进行。

怎么做　提高代码的可扩展性

要想让代码具有灵活性，就要保证代码易于扩展且扩展时不会波及无关的元素。

不过，灵活性是一把双刃剑。代码易于修改自然是好事，但灵活性往往会成为将复杂的代码或设计正当化的借口。不能以牺牲简洁性为前

提来提高灵活性，因为多余的灵活性是无用的，为这种灵活性服务的代码到头来很可能成为一堆带来多余复杂性的无用代码。

为了写出灵活且简洁的代码，我们需要克制住展现小技巧的欲望。相较于通过设计自上而下地获取灵活性，从确保简洁出发，通过单元测试自下而上地获取灵活性会更好。



实现编程理论的六个原则①

效应局部化

英 语 Local consequences

是什么　减少修改带来的影响

效应局部化中的“效应”是指修改带来的影响。

效应局部化是指将修改带来的影响控制在局部。

效应局部化是一个很重要的原则。围绕该原则产生了许多技术，模块化就是其中之一。模块化技术的目标之一就是让修改模块所带来的影响停留在该模块的内部。

为什么　更易于修改和确认

在效应非局部化的情况下，某处修改会对其他完全不相关的地方造成影响，使修改成本大幅增加。

这时如果知道哪些地方受到了影响，或许还有救。然而在大部分情况下，我们对此是一无所知的。这时就得先花时间排查影响范围。

但在效应局部化的情况下，我们需要阅读的代码以及修改所带来的影响都会限制在一定的范围内。

另外，效应局部化还有让交流更加顺畅的效果。在效应局部化的情况下，程序员只要理解当前阶段所涉及的代码即可，不需要一次性掌握所有代码。

怎么做　整合关系紧密的代码

在编写代码时，要让关系紧密的代码集中在一起，同时保证关联性较弱的代码不互相依赖。为此，我们需要将关系紧密的代码集中起来实现模块化。

从关联性的角度来看，我们要格外注意相互频繁调用的模块。模块之间相互频繁调用通常表明原本应该放在一起的要素被分别放在了不同的模块中。

这种时候就需要让合适的单一模块来实现功能，或者将模块功能整合在一起，或者重新创建一个模块来实现相应的功能，总之要保证单一功能的密集性。



实现编程理论的六个原则②

3.6 重复最少化

英　语　Minimize repetition

是什么　消除重复

重复最少化，就是指极力消除重复。

许多技术都以实现重复最少化为目标，函数化技术就是其中之一。该技术将重复的逻辑函数化，整合成一段共享代码来使用。

为什么　将修改带来的影响局部化

重复的代码不符合效应局部化原则，增加了修改成本。

比如，当通过复制粘贴让同一段代码出现在多个位置时，如果有一个地方进行了修改，我们就必须检查其他地方，判断各处是否需要修改。这个“判断”很难把握，并不是全部替换就万事大吉了。而且检查时不能只看复制的部分，其周围的部分也要检查，只有这么做才能正确地进行修改。

这一过程的成本非常高。

怎么做　对代码进行分割管理

我们要将代码分割成多个小块。

大块代码之间一般会存在重复的部分。将大块代码分割成多个小块之后，就能轻松找出共同的部分了。

将代码分割成多个小块后，要明确代码内哪些地方是完全一样的，哪些地方是相似的，哪些地方是完全不同的。只要区分清楚这些，代码的可读性就会提高，修改成本也会下降。



实现编程理论的六个原则③

3.7 逻辑与数据的一体化

英　语 Logic and Data together

是什么　将数据与逻辑放在相近的位置

逻辑与数据的一体化是指把逻辑和该逻辑处理的数据放在相近的位置。

所谓相近的位置，指的是在同一函数或同一模块内。距离越近，代码的质量就越高。

为什么　数据与逻辑往往需要同时修改

修改代码时往往需要同时修改逻辑与该逻辑处理的数据。

因此，如果把二者放在同一位置，我们要阅读的代码就会减少，修改也不会波及其他元素。从结果来看，这么做降低了修改成本。

怎么做　将数据与逻辑放在相近的位置

我们要把数据与逻辑放在相近的位置。

不过，我们很难一开始就知道哪个逻辑应该和哪些数据放在一起。这时不妨先大致安排一下，之后再根据具体情况进行调整。

编写、运行代码之后，数据与逻辑的关联性会渐渐显露出来。是让代码靠近数据，还是让数据靠近代码；是将代码和数据取出来放到其他位置，还是不进行任何变动……对于这些问题，我们会在不断尝试的过程中逐渐了解具体的做法。



实现编程理论的六个原则④

对称性

英 语 Symmetry

是什么 让代码具有一贯性

对称性一般指事物中对某种变换保持不变的性质。以图形中的对称性来说，“轴对称”就是“镜面对称”，“点对称”就是“旋转对称”，这些都不难理解。

而编程中的对称性要比图形中的对称性抽象得多。编程中的对称性是指相同的思路在代码的任何地方都以相同的形式表现出来。

简单来说，就是组内的等级整理。同类的东西，也就是拥有相同性质的东西，要使用相同的等级来表现。

为什么 可以类推其他部分

在代码中明确表现出对称性后，代码的可读性将大幅提高。

对读代码的人而言，具有对称性的代码要比参差不齐的代码更容易理解，读代码的人还可以通过阅读部分代码来类推其他部分。

对称性存在于自然界、数学和艺术等方方面面，给我们带来美丽、舒适的视觉感受。对称性虽然只是一种视觉感受，但它对代码的可读性有着巨大的贡献。

另外，追求对称性从本质上来说属于消除重复代码的准备工作。当代码中存在多个相似的部分时，如果对其进行对称化处理，相同的思路就会以相同的形式表现出来。此后只要将完全相同的部分整合到一起，就可以消除代码中重复的内容了。

怎么做　相同的东西用相同的形式表现

在编写代码的时候，同类的东西要用相同的形式表现。具体如下所示。

- 如果有“添加”方法，就创建与之相对的“删除”方法
- 同组的函数用相同的参数
- 让同一个模块里的数据拥有相同的生存周期
- 函数内所有调用函数的抽象级别要相同



实现编程理论的六个原则⑤

声明式表达

英　语　Declarative Expression

是什么　声明式编程

声明式表达是指在表达代码意图时，尽量用“声明式”的表达方式，而非“命令式”的表达方式。

命令式编程描述的是问题的解决方法，也就是数据结构与算法。而声明式编程描述的是问题的定义，也就是当前问题的性质及解决问题时应满足的限制条件。

为什么　没有流程方面的限制，可读性更高

声明式的代码没有流程方面的限制。这种单纯阐述事实的表达方式能够提升代码的可读性。

另一方面，我们要想正确理解命令式的代码，就必须时常在脑中描绘其状态、控制及数据流。为此，我们必须跟着代码的流程走向来进行阅读。

怎么做　采用声明式的表达方式

我们要采用声明式的表达方式，简洁地表达意图。

当编程范式使用了声明式语言时，由于编写出的代码是声明式的，所以不用我们去特别注意什么。比如函数式语言就是声明式通用编程语

言中的一个典型代表。另外，HTML、CSS 和 SQL 等非通用语言也都是声明式的。

另一方面，当编程范式使用了命令式语言时，我们也要在代码中合适的部分使用声明式表达，以获取声明式带来的优势。代表方法有注释和 DSL（Domain Specific Language，领域特定语言）。



实现编程理论的六个原则⑥

变动率

英 语 Rate of Change

是什么 按修改理由进行分组

变动率体现了修改代码的时间点，变动率相同意味着代码在同一时间点被修改。

同时修改的元素要放在同一个地方，在不同时间点修改的元素要放在不同的地方。

这可以说是对时间应用了对称性原则。同样的东西要同等对待，这一点对修改代码的时间来说也一样。同一时间修改的东西要放在同一个地方。

为什么 能缩小修改范围

在模块等按照某种单位分组的代码中，有多个修改理由的代码比较脆弱，因为一个模块中如果修改代码的时间不同步，那么修改带来的影响将会波及无关的部分。修改理由越多，修改的机会就越多。这种代码一般会承担多项职责，所以代码量较大。经过修改之后，代码量会进一步增加。久而久之，这类代码就会像建在沙地上的楼阁一般，摇摇欲坠。

相反，如果模块只存在一个修改理由，就代表该模块由关联性极强的代码集合而成。这类模块满足高聚合性，非常牢固。因为修改范围较小，影响范围也小，所以模块修改起来比较轻松。

怎么做　根据修改理由分配位置

修改时间相同的元素要放在同一个地方，修改时间不同的元素要放在不同的地方。这对逻辑和数据来说都适用。

先看逻辑。举个例子，如果税额计算包括“一般计算逻辑”和“各年固有逻辑”两种，这两种逻辑就要分别放在不同的地方，因为二者的修改时间明显不同。这样一来，每年在修改“各年固有逻辑”时，就能保证“一般计算逻辑”不受影响。

再看一下数据。比如模块中的一部分数据只在执行某个特定函数时才使用，这时就要将这部分数据移至函数中，把它们用作本地变量。

另外，即使某个模块中存在一些同时被修改的数据，如果这些数据不存在与其他变量同步修改的必然性，我们也要把这些数据移至别的（辅助）模块中。假设有一个用于对金融商品进行计算的“金融商品”模块，该模块中有“通货”和“金额”两组数据。我们新建一个“货币”模块，将“通货”和“金额”的数据移动过去。这样一来，汇率换算和添加通货种类等处理都可以被“货币”模块吸收，“金融商品”模块就能专注于金融商品的计算了。

关联信息　单一职责原则

“单一职责原则”（the Single Responsibility Principle，SRP）规定一个模块只能有一个修改理由。

有多个修改理由就意味着模块承担了多项职责。我们不可以创建这样的模块。

之所以这么说，是因为承担多项职责的模块十分脆弱。当一个模块承担了多项职责时，只要其中一项职责被修改，其他不相关的地方就会受到影响，模块很可能在不经意间遭到破坏。在维护过程中，如果负责修改的程序员不是当初编写代码的人，那么这个人在修改代码的时候很可能注意不到模块承担了多项职责。这样一来，模块遭到破坏的风险就会进一步加大。

模块的某个部分有多个修改理由就证明该模块违反了单一职责原则。一个模块只承担一项职责，模块必须专注于某项职责 。

变动率原则可以帮助我们实现这种状态。将变动率相同的代码放在一起，自然而然就能形成满足单一职责原则的模块。



3.11 软件架构基本技法

英 语 Enabling Techniques for Architecture

是什么 优质代码的基本原理

软件架构基本技法共有以下 10 种。

- 抽象
- 封装
- 信息隐藏
- 打包
- 关注点分离
- 充足性、完整性、原始性
- 策略和实现的分离
- 接口与实现的分离
- 单一引用点
- 分治

为什么 优质代码都有“型”

开发优质的软件需要有基本技法作为支撑。

这些基本技法就好比空手道中的“型”。这些“型”源于软件开发历史中无数程序员积累而来的实践经验。

程序员认识到，对于一个问题，特定的解决方案要优于其他方案，于是这些解决方案被重复使用。这些方案就是基本技法。

怎么做 掌握“型”

我们要把基本技法应用到代码之中。

基本技法不是从某种软件开发技术中总结出来的。这些技法是更为本质的东西，它们适用于一切开发方法以及编程语言。

笔者会在接下来的几节对各个基本技法进行详细说明。

出处

[1] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal. Pattern-oriented software architecture[M]. New York: Wiley, 1996.

相关图书

[1] 細谷功. 具体と抽象—世界が変わって見える知性のしくみ [M]. 東京: dZERO, 2014.

[2] 千叶滋. アスペクト指向入門—Java・オブジェクト指向から AspectJ プログラミングへ [M]. 東京: 技術評論社，2005.

[3] 克雷・拉蒙. UML 和模式应用 [M]. 李洋，郑龚，译. 北京: 机械工业出版社，2006.

[4] 艾伦・沙洛维，詹姆斯・ R. 特罗特. 设计模式解析 [M]. 北京: 人民邮电出版社，2010.

[5] 蒂莫西・巴德. 面向对象编程导论 [M]. 黄明军，李桂杰，译. 北京: 机械工业出版社，2003.

[6] 株式会社テクノロジックアート . オブジェクト指向プログラマが次に読む本—Scala で学ぶ関数脳入門 [M]. 東京: 技術評論社，2010.

[7] 格雷迪・布奇，等. 面向对象分析与设计 [M]. 王海鹏，潘加宇，译. 北京: 人民邮电出版社，2009.



软件架构基本技法①

抽象

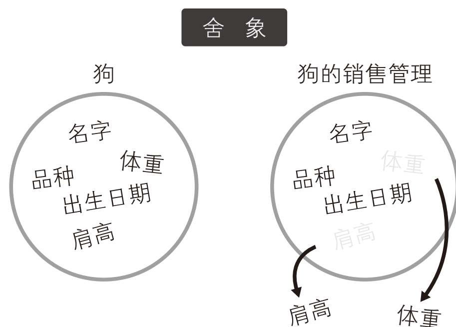
英 语 Abstraction

是什么 在概念上“划清界限”

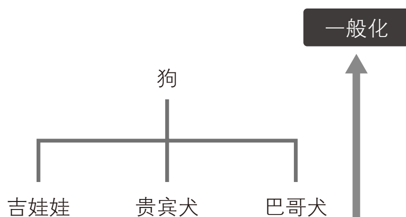
抽象，是指在概念上明确划清界限。这种明确划清界限的方式可以将一个模块与其他模块区分开来。

抽象由“舍象”和“一般化”这两个观点组合而成。

舍象指的是舍去复杂对象的几个性质，只关注其特定的性质。



一般化指的是从具体对象中抽出共同的性质，将其固定为更加通用的概念。



为什么　抽象是对抗复杂的手段

抽象是人们在处理复杂事物时使用的基本手段。

舍象可以除去对象的枝叶，让它的本质暴露出来。复杂度下降之后，我们便能集中精力去对付真正的问题。抽象化的概念简明扼要，便于使用，应用范围极广。

一般化通过抽取共同特征将多个对象聚成一组，将它们视为相同的事物。如此一来，从一个事物中学到的东西就可以应用到其他事物中，也就是所谓的闻一知十。

编程虽然没有特效药和万能药，但在解决问题方面，抽象化思维是一个有力的武器，能帮我们引导出更好的、更有效率的解决方案。

怎么做　使用"舍象"和"一般化"

我们要对事物进行抽象化处理。

抽象化是程序员实现优秀设计的基本技术。

在处理复杂事物时要进行舍象。舍弃多余的东西，抓住本质。需要注意的是，事物的本质固然重要，但在处理问题时，我们更应该关注当前问题的本质。

在处理多个不同的事物时要采用一般化的手段。抽取对象共同的性质，将这些共同点组合成通用的概念。本书所说的原则就是一个典型的例子。一般化是一种用于在多个事物之间总结原则的技术，是一种模式识别。我们可以从身边的事物中找出模式，为其命名，将其作为原则应用到其他场合中。

另外，抽象的反面是具体。具体的事物便于理解，但一个一个分析具体级别不同的事物会消耗无尽的时间，况且这种具体事物的分析结果并不能应用到其他事物中。

只有经过抽象化处理，大量的对象才能套用统一的公式，因此在思考的效率上，抽象远胜于具体。



软件架构基本技法②

封装

英 语 Encapsulation

是什么 给数据和逻辑分组

将相互关联的数据和逻辑分到一组，定义为一个模块。用模块这层外衣将关联性较强的数据群和逻辑群包裹起来的做法称为封装。

各个模块会作为完全不相关的东西分开进行处理。

为什么 不混淆抽象概念

通过分组，相互关联的元素被集中到一个模块中，共同担负起一个抽象概念。这种做法有以下优点。

- 模块中不存在不相关的元素，代码可读性提高
- 修改带来的影响被限制在模块内
- 影响程度明确，代码更易于修改
- 各个模块都是独立的零件，便于重复使用
- 数据和逻辑分割成了小单位的模块，便于我们处理复杂问题

怎么做 将相关元素封装在一起

将相互关联的数据和逻辑分到一组，整理成一个模块。

封装时仅添加相关元素，决不能让无关的元素混入模块中。有了关联性很强的数据组成的数据结构，以及关联性很强的逻辑组成的函数群之后，我们便能获得简洁且高质量的模块了。



软件架构基本技法③

信息隐藏

英 语 Information Hiding

是什么 只展示必要的信息

对使用模块的用户隐藏模块实现相关的信息。

模块内有哪些数据，函数是用何种逻辑实现功能的，这些信息全部对外隐藏。我们要让外部无法直接访问模块内的数据。模块的函数也尽量不公开。

用户只能通过最低限度公开的函数来操作模块。

为什么 整理关系以达到简洁

将用户不必知晓的内部详细信息隐藏起来，可以减少接口的代码量，让信息交互变得更简洁，降低代码的复杂程度。

从用户的角度来看，由于排除了多余信息的干扰，模块的使用方法变得简单，模块也变得更加好用了。

另外，公开的部分越少，模块内部的修改就越不容易影响到外部。这样可以将修改代码的影响范围控制到最小。

怎么做 隐藏内部信息

仅公开模块简单的功能，内部状态和内部功能全部隐藏。禁止外部访问模块内部的数据以及仅供模块内部使用的函数。

实现信息隐藏需要使用封装的手段。分组可以整理“关系网”，降低隐藏信息的难度。

扩展　封装与信息隐藏的区别

封装与信息隐藏是两个不同的概念。我们再来看一下二者的定义。

- **封装**

  将相关元素集中起来模块化。

  将关系紧密的数据与函数整合在一起。

- **信息隐藏**

  隐藏模块的内部状态和内部函数。

  阻止外部对内部的直接访问。

不过，很多技术文档将信息隐藏归到了封装中，可能是因为“封装”二字给人一种密封、隐藏的感觉吧。

这种被扩充了语义的“封装”一词被广泛使用，所以大家在遇到这个词语时，要仔细阅读前后文，注意其真正的含义。

关联信息　Parnas 原则

Parnas 原则是面向对象编程中使用的原则。以下两个条件定义了该原则。

- 对于模块的使用者，仅提供使用该模块所必需的所有信息，其余信息一概不予提供
- 对于模块的开发者，仅提供实现该模块所必需的所有信息，其余信息一概不予提供

模块之间的关系越简单越好。这样我们就能获得两个优势：一是即使不知道模块的内部信息，也可以使用模块（重复利用）；二是可以轻松替换模块的内部实现，而不必担心对使用者造成影响（维护）。



软件架构基本技法④

打包

英 语 Packaging

是什么 给模块分组

将模块按照某种有意义的单位整理并分组，其实就是将整个软件按照某种有意义的单位进行分割。这种分割后的产物称为“包”。

包决定了以何种方法将软件的理论结构存储为物理结构。包起到的是物理容器的作用，主要用来以某种有意义的单位存放软件的功能。

为什么 降低模块群的复杂度

将代码中的相关元素封装成模块，可以起到整理代码、降低复杂度的作用。然而，当软件规模大到一定程度之后，模块的数量也会变得非常多，这同样会增加复杂度。此时就需要对模块群进行分组，也就是打包。打包有以下优点。

- 整个软件被分割成包，复杂度下降
- 包内没有不相关的模块，便于管理
- 基本可以把修改带来的影响限制在包内，代码的修改因此变得更加方便
- 依赖关系得到整理，方便代码以包为单位重复使用

怎么做　自下而上地设计包

将相互关联的模块集中起来打包。

我们要等模块积攒到一定数量之后再自下而上地对包进行设计。这项工作不可能一开始就通过自上而下的方式来完成。

包这种比较大的分解单位容易被误认为是用于功能分解的单位，但包其实是表现软件构建方法的“图纸”。项目最初并没有可以构建的软件，所以不可能实现自上而下的设计。

随着开发的进行，模块将越来越多。此时我们就可以开始自下而上地设计包了。包的设计并不是一锤子买卖，包还要随着编程的推进不断成长和进化。



软件架构基本技法⑤

关注点分离

英 语 Separation of Concerns

是什么 根据关注点分离代码

关注点指软件的功能或目的。把关注点“分离”，就是将与各个关注点有关的代码集中起来做成独立的模块，与其他代码分离。分离后的模块要尽量减少公开的功能数量，与其他模块的关联也要维持在最低限度。

在设计技法中，有很多模式用于实现关注点分离。其中最具代表性的模式是“模型–视图–控制器”（Model-View-Controller，MVC）。在编程领域，关注点分离的代表技术是面向切面编程（Aspect-Oriented Programming，AOP）。

为什么 修改以关注点为单位

代码的修改通常以关注点为单位。因此，将代码按照关注点进行分离有以下好处。

- 各个关注点互相独立，从而缩小了代码的修改范围，使代码更易于修改
- 修改带来的影响限制在了关注点之内，因此代码的质量能够保持稳定
- 因为代码的编写是以关注点为单位进行的，所以能够实现并行开发

怎么做　以关注点为单位模块化

以关注点为单位创建模块，把不同的功能、不相关的功能分开。比如在“模型–视图–控制器”模式下，业务逻辑、用户显示和输入处理互相分离。

另外，如果一个模块在不同前提下有不同的功能，我们就需要分割该模块，让各个功能处于独立的状态。一般来讲，一个模块不可以拥有多个功能。

关联信息　面向切面编程

面向切面编程是一种擅长分离“横切关注点”的技术。所谓横切关注点，就是指横穿于各个关注点之间的关注点。

一般情况下，横切关注点会散布在各个功能之中。这是妨碍我们修改代码的一大因素。横穿所有功能的日志功能就是一个很好的例子。日志功能虽然由独立的模块提供，但日志的调用会直接写在各个功能之中。

除了日志功能，数据库事务的开始及结束处理、防火墙的访问控制处理等也具有横切关注点的特征。

在面向切面编程中，这些横切关注点会根据某种结合规则自动添加到各关注点的相应位置，从而避免被代码直接调用。面向切面编程就是通过这种方式实现关注点分离的。



软件架构基本技法⑥

3.17 充足性、完整性、原始性

英 语 Sufficiency, Completeness, Primitiveness

是什么 表达要充足、完备且精练

经过封装，我们让相互关联的元素集中到一个模块来承载一个抽象概念。该模块承载的抽象概念要具有充足性、完整性和原始性。

- **充足性**

  充足性是指模块表达的抽象概念非常充足。

  比如模块要表达“收集”这一概念，但只提供了 remove，没有提供 add，这就不足以表达“收集”这一概念了。

- **完整性**

  完整性是指模块表达的抽象概念具有所有特征。涵盖所有特征、没有缺漏的模块方便任何人拿去使用。

  比如当模块要表达“收集”这一概念时，如果模块没有提供用于获取元素个数的 size，我们就不能说该抽象概念具有完整性。

- **原始性**

  原始性是指模块表达的抽象概念非常精练。

  比如当模块要表达“收集”这一概念时，如果已经提供了表示添加 1 个物品的 add，则不需要再提供表示添加 10 个物品的 add10。从抽象的精练性这一角度来看，这么做是多余的。

为什么　准确表达抽象概念

模块表达的抽象概念必须能向使用者传递有用的信息，准确表达意图。

如果模块不具备充足性，信息就会出现缺失，导致用户看不清模块的本质。人们无法把说不清道不明的模块拿来使用。

不具备完整性的模块无法让用户安心使用。一旦出现功能不足的情况，工作就难以进行下去。

不具备原始性的模块用起来很麻烦。复杂的接口提升了理解的难度，让人弄不清楚使用时机和使用方法。

怎么做　完美表达模块的抽象概念

我们要明确模块要表达的抽象概念。

信息过多或过少都会使信息的传达变得不准确。因此模块提供的函数要满足充足性、完整性和原始性。当出现多余的内容时，要么将其删除，要么将其移至其他模块。



软件架构基本技法⑦

策略和实现的分离

英 语 Separation of Policy and Implementation

是什么 策略和实现不同时存在

模块可以承载策略或实现。但是，一个模块不可以同时承载二者。

- **策略模块**

  策略模块依赖于软件的前提条件，这类模块通常用于给业务逻辑或其他模块选择参数。

- **实现模块**

  实现模块不依赖于软件的前提条件，这类模块通常是独立的逻辑部分。软件的前提条件会作为传递给模块的参数给出。

为什么 实现稳定但策略不稳定

实现模块不依赖于特定的软件，是一种纯粹的模块，因此可供其他软件重复使用。

而策略模块是为特定软件量身打造的。当软件发生变动时，策略模块也要被迫发生改变。因此，如果将实现和策略混在一起，那么当策略发生变动时，实现也会受到牵连，影响模块的重复使用。

怎么做　将策略与实现存放在不同的模块中

设计时要有意识地区分依赖于软件前提条件的策略部分和不依赖于软件前提条件的实现部分，然后将二者分别写入不同的模块。

当遇到无法分离的情况时，至少要在模块内部明确区分策略部分和实现部分。

另外，即便最初分离了策略与实现，在维护代码的过程中二者也可能会重新混在一起。因此在开展修改工作之前，要明确修改对象到底属于策略还是实现，至少保证实现部分中不会混入策略的相关代码。



软件架构基本技法⑧

接口与实现的分离

英 语 Separation of Interface and Implementation

是什么 模块由接口和实现组成

模块由接口和实现这两个相互分离的部分组成。

- **接口部分**

  接口部分用于定义模块具备的功能，决定模块的使用方法。该部分由用户可访问的函数签名组成。

- **实现部分**

  实现部分其实就是实现模块功能的代码部分。该部分包含模块内部使用的逻辑和数据。

  用户无法访问实现部分。

为什么 使用者只需理解接口

接口与实现分离之后，模块的使用者（用户）就不必了解实现的详细内容了。这样一来，展现在用户眼前的就只有接口，模块变得简单易懂，可以让用户轻松使用。

另外，接口与实现分离能保证“模块的使用方法”（=接口）和“功能的实现方法”（=实现）的独立性。如此一来，修改实现部分时就不必担心会对接口造成影响了。

怎么做 针对接口编程

关于模块的设计原理，有一句“针对接口编程，而不是针对实现编程”的格言。因此，模块之间的调用要保证只使用接口来完成。接口的实现要隐藏在接口背后，该部分不能被直接调用。



软件架构基本技法⑨

3.20 单一引用点

英 语 Single Point of Reference

是什么 只定义一次

模块的各元素仅被声明和定义一次。

以变量为例，仅定义一次就是指初始化之后就不再对值进行更改了。这样一来，我们就不用再追踪变量值的变迁过程，代码的可读性也随之提高。

为什么 使编程无副作用

该基本技法可以使编程无副作用。

所谓编程的副作用，是指某一功能使模块状态产生变化，对此后得到的结果造成影响。排除副作用能免去状态的变化，保证相同数据能始终获得相同的结果，从而减少状态依赖引发的故障。

另外，当某一变量被大范围使用且该变量的值被频繁更改时，我们将很难对代码进行追踪。变量值固定则可以省去这些麻烦，从而提高代码的可读性。

怎么做 单一赋值

所谓单一赋值，是指仅对变量执行一次赋值操作。我们要将副作用视为“应该避免的东西”，把变量看作“不变的东西”，给变量赋值后就不再对其进行更改。

大多数编程语言允许对变量重新赋值，但是我们要跳出语法的限制，通过使用常量、设置规则来禁止重新赋值等方式，编写没有副作用的代码。

扩展　通过控制变量提升代码质量

即便我们想努力实现单一引用点这一基本技法，但在实际编程中还是要管理大批变量，这就难免会发生故障。

容易发生故障的都是那些非必要的可变变量。如果消除不必要的可变性，增加不可变变量，代码的质量就能得到提升。

同时，对于剩下的（必要的）可变变量，要尽量减少访问它的逻辑数量，缩小其作用域。我们要养成写小函数的习惯，限制各个函数的职能。要保证各个函数只作用于传递给它们的参数。

如此一来，即使我们为变量赋了非法的值，也能轻松找出问题所在。反过来，如果代码不满足上述条件，我们就需要推测非法赋值是在什么情况下发生的，这样就会在排查故障方面耗费大量时间。

关联信息　引用透明性

引用透明性指函数拥有以下两种特性。

- **调用结果只依赖于参数**

  这个特性说的是数学中的函数。

  给参数传递相同的值总能得到相同的返回值，也就是说，返回值只依赖于参数的值。这种特性称为“单纯性”。

- **调用不影响其他功能的运行**

  这个特性是说函数没有副作用。所谓副作用，是指“某一处理引起状态改变，对此后的处理结果产生影响”。

具有引用透明性的函数独立于外部状态运行，这类函数会在自身内部完成所有处理。

具有这种特性的函数测试起来非常简单。我们只需准备一系列测试数据作为参数值，然后验证函数调用结果即可。

引用透明性对优化处理也有一定好处，比如可以用已知的处理结果代替函数调用等。



软件架构基本技法⑩

3.21 分治

英 语 Divide and Conquer

是什么 将大问题分割成小问题

分治是指将难以直接解决的大问题分割成多个小问题，然后逐个解决。

分割后的小问题要比最初的问题简单许多，比较容易解决。各个小问题被逐一解决之后，原来的大问题就不复存在了。

为什么 大问题无法控制

直接解决大问题的话，难度较大，花费的时间也比较多，闹不好到最后问题也解决不了。之所以会出现这样的现象，是因为规模过大的问题，其复杂度也比较高。

先将问题分割成容易控制的大小，再着手解决，这么做效率比较高。

怎么做 将问题分割后逐个击破

先对问题进行分割，然后再一个一个解决。比如下面几种做法。

- 在设计整个软件时，先将软件分割成多个可以独立设计的部分，然后对这些部分一一进行设计
- 在设计模块时，按照职责对模块进行分割

- 在设计算法时，可以像归并排序一样，先自下而上地对问题进行分割，再探讨能否解决问题
- 在处理海量数据时，可以像MapReduce一样，先将计算分割成较小的单位，然后探讨能否把这些计算交给分布式环境并行处理



软件架构的非功能需求

英 语 Non-functional requirement for Architecture

是什么 “功能之外的功能”的观点

非功能需求指功能方面以外的一切需求。

要想让软件具有高质量，真正服务于用户，单纯满足功能需求是不够的，还要满足非功能需求。在软件架构设计中，非功能需求与功能需求一样重要。非功能需求包含以下几种观点。

- 易变性
- 互操作性
- 效率性
- 可靠性
- 可测试性
- 可复用性

为什么 非功能需求在软件发布后具有很大的影响力

非功能需求对开发、运维以及计算机资源的高效运用有着很大的影响。另外，在发布后的运维阶段，比较大的问题多是由性能、系统宕机等非功能需求引起的。

然而与功能需求相比，这些重要的非功能需求往往被忽视、推后。其实，非功能需求应该在开发最初，也就是设计软件架构的时候就纳入考量。

怎么做　从非功能需求的观点进行设计

在软件架构的设计阶段就将非功能需求纳入考量，绝对不能拖到后面再做。

具体做法如下。

- 在需求定义方面，确认各个观点的被需求程度
- 在开发方面，从软件架构的设计阶段开始就将非功能需求纳入结构之中
- 在测试方面，确认需求是否得到满足

从 3.23 节开始，笔者将逐一讲解非功能需求的各个观点。

扩展　非功能测试

功能测试着眼于“做什么”，而非功能测试着眼于“怎样运作”。二者的着眼点不同。

非功能测试与功能测试同样重要。软件的目的不是实现功能，而是让用户达到自己的目的。其中，非功能需求占了很大的比重。

要想满足非功能需求，首先要找出适合当前软件的非功能需求标准，然后给非功能测试设置一个合格线，毕竟没有目标就无法满足需求。

关联信息一　非功能安全性需求

在非功能需求中，“安全性”（security）非常重要。

简单来说，信息的安全性就是指保护软件所涉及的信息资产不被非法访问、泄露和篡改。随着以使用网络为前提的软件越来越多，用户对安全性的要求也变得越来越高。

信息安全性的定义是维持信息的机密性、完整性和可用性。人们将机密性（confidentiality）、完整性（integrity）和可用性（availability）

称为“信息安全三要素”。取这三个单词的首字母，信息安全三要素又叫作“信息安全的 CIA”。

- **机密性**

  机密性指对未获授权的个人、实体或进程不公开信息或禁止其使用信息。也就是不让未获得许可的人或其他对象使用信息。

  如果第三者能轻松访问重要信息，就会有信息泄露的风险。因此，我们需要给重要的信息添加访问控制，也可以给信息加密，这样一来，即使信息泄露，其他人也无法阅读信息内容。

- **完整性**

  完整性指确保信息正确、完整。

  如果第三者擅自改写、删除重要的信息，这些信息就会失去使用价值。不仅如此，一旦使用者没有发现信息遭到篡改，问题的严重性就可能会进一步加大。

  因此，人们通常使用数字签名等技术来确保完整性。

- **可用性**

  可用性指当获得授权的实体发来请求时，允许实体访问和使用信息。也就是说，实体可以随时访问和使用信息。对组织而言，能够随时访问自己所持有的重要信息这一点非常重要。

  因系统故障而无法访问信息，或者计算机病毒导致系统无法使用等，都属于重大问题。

  因此，我们需要通过对系统和信息进行备份、设置防火墙抵御病毒等手段来防范问题发生。

关联信息二　检验安全性

在检验安全性的时候，首先要进行“渗透测试”(penetration test)。渗透测试是一种为寻找安全漏洞而对软件进行攻击并尝试入侵的检验方法。

不过，这种检验方法需要一些高级知识和高超的技术水平作为支撑。在检验时，借助漏洞攻击工具，可以减少检验项目的遗漏。另外，外包给专门的组织，由外部来进行安全性诊断也是一个不错的方法。这种做法虽然会提高成本，但可以得到专家的验证，能让用户更放心地使用软件。

此外，在检验安全性时，我们还要考察发生安全事故时软件的追踪能力和监察能力。从这一角度出发，访问日志的内容、保存时间等也需要检验。

在思考安全性的相关对策时还要注意一点，那就是软件的易用性。如果软件的用户认证需要极长的密码，或者用户需要进行大量的操作才能获取信息，用户理解和操作软件的难度就会加大。所以，我们在检验安全性需求的同时也要把握其与易用性之间的平衡。

出处

[1] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal. Pattern-oriented software architecture[M]. New York: Wiley, 1996.

相关图书

[1] 迈克尔・C. 费瑟. 修改代码的艺术 [M]. 侯伯薇，译. 北京：机械工业出版社，2014.

[2] 日本情報システム・ユーザー協会. 非機能要求仕様定義ガイドライン [M]. 東京：日本情報システム・ユーザー協会，2008.

[3] 罗伯特・L. 格拉斯. 软件工程的事实与谬误 [M]. 严亚军，龚波，译. 北京：中国电力出版社，2006.

[4] 埃里克・埃文斯. 领域驱动设计 [M]. 赵俐，盛海艳，刘霞，译. 北京：人民邮电出版社，2010.

[5] 岸本了造. テクニカルエンジニア 情報セキュリティ標準教本 [M]. 東京：日本経済新聞社，2007.



软件架构的非功能需求①

易变性

英 语 Changeability

是什么 轻松修改软件的能力

易变性指软件能被轻松改良的能力。

具体来讲，易变性要求软件能被轻易修改、轻易扩展、轻易重组、轻易移植到其他平台。此外，在保证质量的前提下迅速完成改良也是易变性的要求之一。

为什么 软件的寿命比预想的长

我们总希望软件在发布之后一直不用修改，但事实并非如此。软件在使用周期内会不断被修改和扩展。原需求会不断发生变更，新需求也会不断添加进来。

特别是业务领域中使用的大规模软件，这类软件通常有着很长的寿命，某些软件甚至能存活几十年。

要想长期快速应对用户的各种琐碎要求，我们就需要设计出便于修改代码的软件架构。

怎么做 可维护性、可扩展性、重组和可移植性

软件架构的设计要从可维护性、可扩展性、重组和可移植性这四个方面进行考察。

- **可维护性**

  可维护性指解决问题的难易度，即修改故障代码的难易度。

  要提高可维护性，使用的软件架构就必须能将修改控制在局部范围内，把对其他模块的副作用降至最低。

- **可扩展性**

  可扩展性指添加新功能、更新模块版本以及删除不需要的功能和模块等操作的难易度。

  提高可扩展性需要降低模块之间的结合度。

  可扩展性追求的是能在不给客户端造成影响的情况下完成模块交换的结构，以及能将新模块整合到现有软件架构中的结构。

- **重组**

  重组指重新组织模块间的关系。当对模块的位置进行调整时（比如将模块移至其他子系统），就需要进行重组。

  为方便重组，我们设计的软件架构要能在不影响模块实现的前提下灵活对模块进行配置。

- **可移植性**

  可移植性指将软件移植至各种硬件平台、用户接口、操作系统、编程语言和编译器等的难易度。

  要提高可移植性，就要在设计软件时考虑软件对硬件的依赖性。

  要想做出不依赖于硬件的软件，就要将系统库和用户接口库等对平台特有功能进行操作的部分整合成一个独立的专用模块。

扩展一　灵活性的取舍

设计优质软件架构的关键点在于，要清楚软件的哪些部分需要具备较高的灵活性以应对修改，哪些部分不会被修改。

在此基础上，对需要具备较高灵活性的部分套用支持修改的设计模式，提高灵活性。

不过，这种支持修改的设计有一点需要我们注意。在选择高灵活性的软件构架时，人们往往把灵活性设计得过高。这样一来，模块就会失去简洁性，导致软件的可扩展性下降。

因此，灵活性不是越高越好。我们要考虑灵活性与简洁性之间的平衡。

扩展二 软件老化

就像人会变老一样，软件也会变老，这就是“软件老化”的观点。

软件老化是指软件随着时间的流逝慢慢退化。软件老化的原因有以下几点。

- 设计灵活性不足，代码的修改使软件架构遭到破坏
- 设计本身没有问题，但负责修改的人没有理解设计，于是软件架构遭到破坏
- 软件架构难以让人理解，最终被混乱的修改所破坏
- 更新停滞，被时代抛弃，软件越来越陈旧

我们无法阻止软件老化，但是我们能够减缓软件的老化速度。通过了解软件老化的原因，按照一定步骤抑制老化进程，及时弥补损伤，相信我们能看到软件完成使命寿终正寝的那一天。

我们可以采用很多手段来防止软件老化，比如准确进行文档化、修改时注意保护软件架构、认真审查、采用具有灵活性的设计准确预测修改位置等。



软件架构的非功能需求②

互操作性

英 语 Interoperability

是什么 与其他软件交互的能力

互操作性指软件与其他软件交互的能力。互操作性要求不同的软件之间使用同一交换方式来交换数据、读取相同格式的文件、使用相同的协议等，从而实现相互连接的状态。

为什么 软件间协作

软件并不是独立存在的，它是系统的一部分，会频繁地与其他系统或环境发生作用。

能与其他软件协作，就意味着我们可以直接使用既有资产。这样一来，需要我们开发的软件就会减少。

良好的连接性能扩大软件的用途，缩短开发周期，减少成本。

怎么做 选择标准规格

在设计软件架构时，要明确定义需要访问的外部功能以及数据结构。这么做能够提升软件的价值。

另外，在选择协议和数据格式时，要选择业界标准规格。从长远来看，选用标准规格能长期保持软件的价值。

不过，在实际开发中，很多时候开发的软件要迎合一些既有软件，保证与这些软件的互操作性。这时，我们要仔细确认连接方式以满足既有软件的连接要求。在两个既有软件相互连接的情况下，可以考虑导入其他软件作为介质。



软件架构的非功能需求③

3.25 效率性

英 语 Efficiency

是什么 高效使用资源的能力

效率性指软件在运行过程中使用资源发挥性能的能力。

效率性大致分为以下两种。

- **时间效率性**

  时间效率性从时间的角度来定义资源的使用效率。

  时间效率性可以通过一定时间内可以完成的处理数量（通量）、从用户执行输入操作到应答所花费的时间（响应时间）、从用户开始操作到输出所需信息所花费的时间（周转时间）等来衡量。

- **资源效率性**

  资源效率性从计算机资源的角度来定义资源的使用效率。

  资源效率性可以通过CPU占用时间、内存使用量、存储空间占用量和网络传输量等来衡量。

为什么 资源是有限的

资源是有限的，所以软件需要高效地使用资源。资源使用方法不当会使软件的运行变得缓慢，导致用户体验变差。

不过，效率问题并不是使用成体系的算法就能解决的。要想提高效率，就要在软件架构的设计阶段将职责分散到各个模块，并将各个模块适当关联起来。

怎么做　活用资源

我们要合理使用计算机的资源。

所谓合理使用，当然不是说用得越少越好，而是指有效利用既有资源来最大限度地发挥软件性能。节约是必须的，但我们在设计软件架构时也要将资源的灵活使用纳入考虑的范围。

扩展　间接化与效率性的平衡

为了避免各个模块直接关联，我们有时会在模块之间导入“媒介模块”。这种方法称为间接化。

间接化是一种基础且适用范围很广的模块设计方法。有这样一句格言：“计算机科学领域的任何问题都可以通过增加一个间接的中间层来解决。”间接化有助于维持低耦合性，保持较高的可维护性、可扩展性与可复用性。很多设计模式将间接化作为基本思路之一来使用。

不过，在采用间接化这种设计方法时，我们要掌握好它与效率性之间的平衡。之所以这么说，是因为间接化会让处理变得冗长，对效率造成影响。

我们要对软件所需的效率性有清楚的认识，掌握好效率性与可维护性、可扩展性和可复用性之间的平衡，在此基础上使用间接化这一设计方法。



软件架构的非功能需求④

3.26 可靠性

英 语 Reliability

是什么 维持功能的能力

可靠性指软件在异常情况下或在被非法、非常规使用时维持自身功能的能力。

可靠性体现在容错和健壮性这两个方面。

● **容错**

容错指软件发生故障时仍保持正常运行的能力。它保证软件能在异常情况下正常运行，并在内部完成故障的修复工作。

修复完成后，软件需要继续或从头开始执行异常位置的操作。比如分布式系统在发生通信异常时会先暂时切断连接，等问题修复完成后再重新连接，恢复软件的运行。

● **健壮性**

健壮性是保护软件不受非正常使用方式或非法输入影响的能力。具备该能力后，不论什么样的使用方式，软件都能准确迁移至系统定义的状态。

健壮性只保证软件能迁移至系统定义的状态，并不要求软件修复或重新执行引发异常的处理。

为什么　软件对可靠性的需求各不相同

软件对可靠性的需求是不尽相同的。

比如大规模系统或关键的业务系统就不允许出现中断服务的情况，就算缩减功能，也要持续提供服务。

而供个人使用的软件就不需要如此强大的持续性了。相较于在危险状态下继续提供服务，这类软件大多选择保存数据重新启动。

因此，在设计软件架构时，我们要先明确软件对可靠性的需求。

怎么做　冗余化、故障弱化和故障安全

从容错的观点出发，我们可以让软件架构存在内部冗余（双重冗余等）。也可以采用故障弱化的设计，在软件发生故障时缩减软件提供的功能，只保留关键功能，保证处理继续进行。

从健壮性的观点出发，我们可以采用故障安全的设计，在发生故障时剥离发生故障的部分。也可以采用故障保护的设计，保证软件在用户进行了错误操作的情况下也能安全运行，避免故障发生。



软件架构的非功能需求⑤

可测试性

英 语 Testability

是什么 有效进行测试的能力

可测试性指软件有效且高效地进行测试的能力。

“有效地进行测试”指测试有深度且高质量，即通过测试可以全面检测软件的质量。“高效地进行测试”指测试所需成本和劳力较少，即能够花费较少的成本快速地检查软件的质量。

为什么 测试的质量即产品的质量

随着软件体积的增大和软件复杂程度的加深，测试的难度会越来越大，所需成本也会越来越高。因此我们要求软件架构不仅要保证软件正常运行，还要有简化测试的效果。

无论是开发阶段还是维护阶段，保证修改后软件的质量都是一个非常重大的课题。在软件的各项需求中，可测试性的重要程度相对较高。

简化软件的测试需要有软件架构的支持。使用能够简化测试的软件架构不仅方便调试代码和调试模块的临时整合，还可以提高排查及修复故障的效率。

怎么做 在设计产品时兼顾测试

从软件架构的设计阶段开始，我们就要将测试方法纳入考虑的范围。

测试代码通常给人一种正式代码附属物的印象，但实际上，正式代码中也可以含有测试代码。这虽然与固有价值观相冲突，但从测试的重要程度来看，还是值得我们让步的。

提高可测试性的关键在于消除模块之间的依赖关系。如果存在依赖关系，难以测试的部分就会拖整个软件的后腿。我们要尽量消除模块之间的依赖关系，保证测试能够以较小的单位进行。



软件架构的非功能需求⑥

3.28 可复用性

英 语 Reusability

是什么 重复使用与被重复使用的能力

可复用性指软件的整体或其中一部分可以在其他软件的开发过程中重复使用的能力。

可复用性表现在两个方面：一个是重复使用现有代码的软件开发；另一个是以重复使用为目的的软件开发。

- **重复使用现有代码的软件开发**
  重复使用现有代码的软件开发指在开发过程中重复使用项目内的既有模块、以往项目的模块和各种库等。将可重复使用的现有代码直接或变形后整合至正在开发的软件中。

- **以重复使用为目的的软件开发**
  以重复使用为目的的软件开发指在当前软件开发中创造出可供未来项目重复使用的模块。为其他软件提供可重复使用的模块是这类软件开发的目的。

为什么 能不做就不做，提高开发效率

为了提高软件开发的效率和软件质量，我们应该尽量避免从零开发。也就是说，从其他地方借用代码是最好的选择。

重复使用现有代码可以让我们少编写一些代码，降低软件开发的

成本，缩短周期。同时，使用已取得成绩的成熟模块能够提高软件的质量。

怎么做 “插件”软件架构

如果是重复使用现有代码的软件开发，我们就要设计出能作为现有结构或模块插件使用的软件架构。

这么做是为了支持软件合成。软件合成指通过利用现有模块来组成软件。为达到这一目的，我们需要实现能够作为黏合剂使用的模块，从而使现有模块适应当前的开发需求。

如果是以重复使用为目的的软件开发，我们就要设计出能将自给自足的部分从正在开发的软件中分离出来的软件架构。自给自足的部分要能在不进行任何修改的情况下直接被其他系统使用。这部分最好做成可独立构建的模块或包。

关联信息一 重复使用的“三之法则”

在创建可重复使用的模块时，有两个“三之法则”。一个是“三倍难度法则”，另一个是“三种测试法则”。

- **三倍难度法则**

  该法则指开发可重复使用的模块的难度是开发在单一软件中使用的模块的难度的三倍。

  在开发可重复使用的模块时，程序员需要考虑一般化问题的处理。除了模块自身要做到一般化，模块的测试也必须能在普通实例中使用。

  因此，模块的可复用性越高，复杂度就越高，对想象力的要求也就越高。整个过程的难度陡然提升。在设计方面，我们必须考虑“什么是一般化问题”；在编程和测试方面，我们还需要考虑“如何处理一般化问题”。

- **三种测试法则**

  该法则指可重复使用的模块在共享之前需要在三个不同的软件中通过测试。

  可重复使用的模块不能满足于解决当前面对的问题，解决更加一般化的问题才是这些模块要达到的目标。

  然而，在开发模块时，我们很难（甚至不可能）将一般化问题想得足够全面，很多问题只有在实践中才会暴露出来。

  因此，我们不可能一开始就做出完美的一般化模块。比较明智的做法是先开发出原型，然后把它放到实际的问题领域中不断完善。

  所以，用于重复使用的模块在开发完成之后，最少要在三个问题领域中进行测试。

关联信息二 可拆装组件的框架

当我们在同一问题领域实现多个软件之后，设计就会愈发成熟，进而突破一些壁垒。此时就可以设计专攻该问题领域的框架了。

框架就好比确定了接口的集线器，问题领域的可变部分是可拆装的组件。框架设计其实就是插件设计。

这是只有极少数在反复开发的过程中获得成功的幸运项目才能达到的理想高度，达到这个高度的软件能够自由地进行修改和扩展。



3.29 七个设计原理

英 语 Seven design principles

是什么 代码有效性审查的观点

七个设计原理是我们在设计代码结构时应考虑的核心观点，目的是避免代码中存在故障隐患。

这些设计原理源于对实际软件开发现场的分析，是提高代码质量的经验结晶。人们从“怎样做才能在开发时避免代码中存在故障隐患”的角度出发，分析一个个故障案例的根本原因，从而总结出了这些原理。

七个设计原理是代码审查的判断标准。这七个设计原理如下所示。

- 简单性原理
- 同构原理
- 对称原理
- 层次原理
- 线性原理（透明原理）
- 清晰原理
- 安全原理

为什么 代码价值观不遗漏且不动摇

代码审查是保证软件质量的有效方法。

不过，审查如果没有一定的价值观或观点作为基础，指出的意见就不会有针对性。在审查者不同或者具体情况不同时，就会出现观点遗

漏、意见前后不一等情况，审查本身的质量都难以保证。因此，我们需要七个设计原理这种固定的判断标准。

怎么做　将七个设计原理应用于代码的编写

将七个设计原理视为代码共同的价值观，用作代码审查的判断标准。七个设计原理也是我们编写代码时应该考虑的内容。

在接下来的几节，笔者将对这七个设计原理逐一进行介绍。

出处

[1] SQuBOK 策定部会. ソフトウェア品質知識体系ガイド—SQuBOK Guide—（第２版）[M]. 東京：オーム社，2014.

相关图书

[1] 乔治・波利亚. 怎样解题 [M]. 涂泓，冯承天，译. 上海：上海科技教育出版社，2007.



七个设计原理①

3.30 简单性原理

英 语 Simplicity Principle

是什么 追求简单

简单性原理就是追求简单。

说得极端一点，就是自始至终都以最简单的逻辑编写代码，让编程初学者一眼就能看懂。

因此，在编程时我们要重视的是局部的完整性，而不是复杂的整体关联性。

为什么 bug 喜欢出现在复杂的地方

软件故障常集中在某一个区域，而这些区域都有一个共同的特点，那就是复杂。编写代码时如果追求简单易懂，代码就很难出现问题。

不过，简单易懂的代码往往给人一种不够专业的感觉。这也是经验老到的程序员喜欢写老练高深的代码的原因。所以我们要有足够的定力来抵挡这种诱惑。

怎么做 编写自然的代码

努力写出自然的代码。放下高超的技巧，坚持用简单的逻辑编写代码。

既然故障集中在代码复杂的区域，那我们只要让代码简单到让故障无处可藏即可。不要盲目地让代码复杂化、臃肿化，要保证代码简洁。



七个设计原理②

同构原理

英 语 Isomorphism Principle

是什么 力求规范

同构原理就是力求规范。

同等对待相同的东西，坚持不搞特殊。同等对待，举例来说就是同一个模块管理的数值全部采用同一单位、公有函数的参数个数统一等。

为什么 不同的东西会更显眼

相同的东西用相同的形式表现能够使不同的东西更加突出。不同的东西往往容易产生 bug。遵循同构原理能让我们更容易嗅出代码的异样，从而找出问题所在。

图表和工业制品在设计上追求平衡之美，在这一点上，同构原理也有着相似之处。统一的代码颇具美感，而美的东西一般更容易让人接受，因此统一的代码有较高的可读性。

怎么做 编写符合规范的代码

我们要让代码符合一定的规范。不过，这会与程序员的自我表现欲相冲突。

为了展现自己的实力，有些程序员会无视编程规范，编写独特的代

码。可靠与简单是代码不可或缺的性质，但这些程序员常常在无意间让代码变得复杂。

这就把智慧与个性用错了地方。小小的自我满足远不及代码质量重要。所以在编写代码时，务必克制住自己的表现欲，以规范为先。



七个设计原理③

对称原理

英 语 Symmetry Principle

是什么 讲究形式上的对称

对称原理就是讲究形式上的对称，比如有上就有下，有左就有右，有主动就有被动。

也就是说，我们在思考一个处理时，也要想到与之成对的处理。比如有给标志位置 1 的处理，就要有给标志位置 0 的处理。

为什么 帮助读代码的人推测后面的代码

具有对称性的代码能够帮助读代码的人推测后面的代码，提高其理解代码的速度。同时，对称性会给代码带来美感，这同样有助于他人理解代码。

此外，设计代码时将对称性纳入考虑的范围能防止我们在思考问题时出现遗漏。如果说代码的条件分支是故障的温床，那么对称性就是思考的框架，能有效阻止条件遗漏。

怎么做 编写有对称性的代码

在出现“条件”的时候，我们要注意它的“反条件”。每个控制条件都存在与之成对的反条件（与指示条件相反的条件）。要注意条件与反条件的统一，保证控制条件具有统一性。

我们还要考虑到例外情况并极力避免其发生。例外情况的特殊性会破坏对称性，成为故障的温床。特殊情况过多意味着需求没有得到整理。此时应重新审视需求，尽量从代码中剔除例外情况。

命名也要讲究对称性。命名时建议使用 set/get、start/stop、begin/end 和 push/pop 等成对的词语。



七个设计原理④

3.33 层次原理

英 语 Hierarchy Principle

是什么 讲究层次

层次原理就是在结构上讲究层次。

注意事物的主从关系、前后关系和本末关系等层次关系，整理事物的关联性。

不同层次各司其职，同种处理不跨越多个层次，这一点非常重要。比如执行了获取资源的处理，那么释放资源的处理就要在相同的层次进行。又比如互斥控制的标志位置 1 和置 0 的处理要在同一层次进行。

为什么 层次结构有助于提高代码的可读性

有明确层次结构的代码能帮助读代码的人抽象理解代码的整体结构。读代码的人可以根据自身需要阅读下一层次的代码，掌握更加详细的信息。

这样一来就可以提高代码的可读性，帮助程序员表达编码意图，降低 bug 发生的概率。

怎么做 编写有抽象层次结构的代码

在编写代码时设计各部分的抽象程度，构建层次结构。保证同一个

层次中的所有代码抽象程度相同。

另外，高层次的代码要通过外部视角描述低层次的代码。这样做能让调用低层次代码的高层次代码更加简单易懂。



七个设计原理⑤

3.34 线性原理

英 语 Linearity Principle

别 名 透明原理

是什么 处理流程尽量走直线

线性原理就是让处理流程尽量走直线。

一个功能如果可以通过多个功能的线性结合来实现，那它的结构就会非常简单。

反过来，用条件分支控制代码、毫无章法地增加状态数等行为会让代码变得难以理解。我们要避免做出这些行为，提高代码的可读性。

“透明”一词可以用来形容代码有较高的可读性，所以线性原理又称为“透明原理”。

为什么 直线处理可提高代码的可读性

复杂的处理流程是故障的温床。

故障多出现在复杂的条件语句和循环语句中。另外，goto 等让流程出现跳跃的语句也是故障的多发地。

如果能让处理由高层次流向低层次，一气呵成，代码的可读性就会大幅提高。与此同时，可维护性也将提高，添加功能等改良工作将变得更加容易。

一般来说，自上而下的处理流程简单明快，易于理解。我们应避开复杂反复的处理流程。

怎么做　尽量不在代码中使用条件分支

尽量减少条件分支的数量，编写能让代码阅读者线性地看完整个处理流程的代码。

为此，我们需要把一些特殊的处理拿到主处理之外。保证处理的统一性，注意处理的流程。记得时不时俯瞰代码整体，检查代码是否存在过于复杂的部分。

另外，对于经过长期维护而变得过于复杂的部分，我们可以考虑对其进行重构。明确且可靠的设计不仅对我们自身有益，还可以给负责维护的人带来方便。



七个设计原理⑥

3.35 清晰原理

英 语 Clarity Principle

是什么 注意逻辑的清晰性

清晰原理就是注意逻辑的清晰性。

逻辑具有清晰性就代表逻辑能清楚证明自身的正确性。也就是说，我们编写的代码要让人一眼就能判断出没有问题。任何不明确的部分都要附有说明。

保证逻辑的清晰性要“不择手段”。在无法用代码证明逻辑正确性的情况下，我们也可以通过写注释、附文档或画图等方法来证明。不过，证明逻辑的正确性是一件麻烦的事，时间一长，人们就会懒得用辅助手段去证明，转而编写逻辑清晰的代码了。

为什么 消除不确定性

代码免不了被人一遍又一遍地阅读，所以代码必须保持较高的可读性。编写代码时如果追求高可读性，我们就不会采用取巧的方式编写代码，编写出的代码会非常自然。

采用取巧的方式编写的代码除了能让计算机运行以外没有任何意义。代码是给人看的，也是由人来修改的，所以我们必须以人为对象来编写代码。

消除代码的不确定性是对自己的作品负责，这么做也可以为后续负责维护的人提供方便。

怎么做　编写逻辑清晰的代码

我们要编写逻辑清晰的代码。

为此，我们应选用直观易懂的逻辑。会给读代码的人带来疑问的部分要么消除，要么加以注释。

另外，我们应使用任何人都能立刻理解且不存在歧义的术语。要特别注意变量名等一定不能没有意义。

扩展一　重复使用代码的风险

如清晰原理所主张的那样，代码必须能让人一眼看出其结构，顺利理解其内容，并且没有歧义。

而重复使用现有代码必须建立在真正理解该段代码的基础之上。

在硬件领域，人们通过有形的实物来学习好的设计。因此，只要留心该实物所处的环境条件，就能实现可以重复使用代码的设计。

而在软件领域就没有这么简单了。软件的环境条件是该软件代码与其他代码相互作用形成的“上下文”。所以，如果看到代码被重复使用过就盲目拿来使用，软件就可能会出现意想不到的故障。

重复使用代码是件好事，但也存在风险。在使用某段代码之前，一定要仔细确认其运行条件和上下文。

扩展二　修复代码故障的风险

相较于重复使用，在修复现有代码的故障时，我们更需要去深入理解代码。

修复必须建立在充分理解代码的基础之上，否则考虑的情况就会不全面，导致代码退化。我们不能只着眼于发生故障的地方。“代码能运行就好”的思想并不适用于修复现有代码的故障。

我们应充分理解代码，充分理解故障类型，进而提出假设，然后在此基础上修复故障，进行全面测试，从而保证代码的质量。



七个设计原理⑦

安全原理

英　语　Safty Principle

是什么　注意安全性

安全原理就是注意安全性，采用相对安全的方法来对具有不确定性的、模糊的部分进行设计和编程。

说得具体一点，就是在编写代码时刻意将不可能的条件考虑进去。比如即便某个 `if` 语句一定成立，我们也要考虑 `else` 语句的情况；即便某个 `case` 语句一定成立，我们也要考虑 `default` 语句的情况；即便某个变量不可能为空，我们也要检查该变量是否为 `NULL`。

为什么　防止故障发展成重大事故

硬件提供的服务必须保证安全，软件也一样。

硬件方面，比如取暖器，为防止倾倒起火，取暖器一般会配有倾倒自动断电装置。同样，设计软件时也需要考虑各种情况，保证软件在各种情况下都能安全地运行。这一做法在持续运营服务和防止数据损坏等方面有着积极的意义。

怎么做　编写安全的代码

选择相对安全的方法对具有不确定性的部分进行设计。列出所有可能的运行情况，确保软件在每种情况下都能安全运行。理解需求和功能，将各种情况正确分解到代码中，这样能有效提高软件安全运行的概率。

为此，我们也要将不可能的条件视为考察对象，对其进行设计和编程。不过，为了统一标准，我们在编写代码前最好规定哪些条件需要写，哪些条件不需要写。

关联信息　代码的必要条件与充分条件

对代码的实现来说，需求和功能的说明书只能算必要条件。

比如说明书里写了需要添加某项数据。然而，当添加数据的操作使数据库发生错误时，说明书并没有明确告诉我们具体的应对措施，比如继续处理然后重试、中止处理并生成日志、让软件停止运行等。

如果只能根据说明书来编写代码，那么能否写出满足充分条件的代码，就得看程序员的本事了。

安全性是满足充分条件必不可少的要素。我们要让代码在满足必要条件的同时准确涵盖各种功能和情况，在与用户保持沟通的过程中，让每个实例都坚实可靠。



UNIX思想

英 语 UNIX Culture

是什么 根植于 UNIX 中的约定俗成的规则

UNIX 思想源于 UNIX 文化。它是大量经验的结晶，是编写优质代码的实用性技术的集合。

UNIX 思想并不是真正意义上的方法论，它是 UNIX 文化孕育出的一系列约定俗成的规则，被人们沿用至今。

这里我们将 UNIX 思想明文化。UNIX 思想可总结为以下几个原则。

- 模块化原则
- 清晰原则
- 组合原则
- 分离原则
- 简单原则
- 简约原则
- 透明性原则
- 健壮性原则
- 表达性原则
- 最小意外原则
- 沉默原则
- 修复原则
- 经济原则
- 生成原则

- 优化原则
- 多样性原则
- 可扩展性原则

为什么 UNIX 设计判断的正确性

UNIX 拥有强大的生命力。

诞生于 1969 年的 UNIX 被人们沿用至今。从信息技术的更新速度来看，这确实是一个傲人的成绩。

另外，UNIX 适用于多种硬件，使用方法也多种多样。

UNIX 之所以如此成功，是因为 UNIX 程序员在设计初期进行了正确的设计判断。这些设计判断得到了全世界优秀开发者的赞同，并被不断完善，于是有了今天的 UNIX。

这些设计判断的精髓便是 UNIX 思想。

怎么做 遵循 UNIX 思想

我们要将 UNIX 思想的各个原则作为设计的方针。这些原则都经过了历史的考验。

虽然 UNIX 思想源于 UNIX 文化，但它在任何软件开发领域都有借鉴意义。接下来的几节笔者将逐一讲解 UNIX 思想的各个原则。

出处

[1] 埃瑞克・S. 理曼德. UNIX 编程艺术 [M]. 姜宏，何源，蔡晓骏，译. 北京：电子工业出版社，2011.



UNIX 思想①

模块化原则

英　语 Rule of Modularity

是什么　以精简的模块为单位

软件是个复杂的东西。不过，软件整体的复杂度是可以降低的。

为此，软件需要由多个简单的模块组装而成。

也就是说，将代码中关联性较强的元素聚集起来做成模块。模块的接口要简单明了，不能含有多余的东西。

为什么　模块间的关系应简洁

接口简明的简单模块不容易与其他模块发生关联。因此，问题能被控制在局部范围，我们可以在不破坏整体代码的情况下修改对象模块。

控制复杂度是编程的精髓。可惜的是，控制复杂度没有捷径可走。我们只能通过精简模块、精简接口来一步一步地降低软件的复杂度。

怎么做　减少模块的接口

我们要尽量减少模块的接口。

模块内部的组成元素之间应具有较强的关联性，这样我们可以在修改某项功能时将影响范围封闭在模块内部。

在此基础之上，我们使用这些简单模块搭建整个软件。



UNIX 思想②

3.39 清晰原则

英 语 Rule of Clarity

是什么 代码要清晰

代码不应该巧妙，而应该清晰。

以大幅提升复杂度为代价，用技术使性能得到一点点提升的做法是丢了西瓜捡芝麻。

复杂的代码不仅难以理解，还容易发生故障。

而清晰的代码既便于理解，又不容易出问题。

为什么 读代码的是人，不是机器

写代码时最重要的一点就是以人（阅读代码维护软件的程序员）为对象，不以执行代码的计算机为对象。

维护是软件不可避免的一个阶段。这个阶段的所需成本通常比重新开发一款软件的成本还要高。因此，代码不是编译完、解释完之后能在计算机上运行就万事大吉了，它还必须能让人读懂。

怎么做 不清晰的地方要进行改善

我们要编写可读性高的、清晰的代码。

另外，在读代码时，不要对难以读懂的部分再三解读。

第一次需要解读可能是因为碰巧没看懂，如果第二次还需要解读，就需要想办法处理了。

此时我们可以给代码添加注释，或者把代码修改得更容易理解，以此来避免再三解读同一段代码。



UNIX 思想③

3.40 组合原则

英　语 Rule of Composition

是什么　软件是能组合的过滤器

开发软件时要保证该软件能和其他软件组合使用。换句话说，就是将软件做成一个尽量简单的过滤器。

这里所说的过滤器是指接收某种数据流之后，经过加工输出另一种数据流的软件。输出的数据流最好是文本格式。

为什么　相互连接可产生协同效应

过滤器形式的软件能轻易地连接在一起。软件通过不同的组合方式，能够完成多种多样的任务。另外，文本流的接口相对简单，这使得软件能够处于信息隐藏的状态。

而无法与其他软件协作的软件通常发挥不出多少价值。即便可以使用特殊的进程间通信方式进行协作，软件也会相互暴露内部结构。

能作为零件使用的软件要具有独立性。具有独立性的软件在与其他软件进行协作时，几乎不必考虑协作方。最理想的情况是一款软件能在不影响协作方的情况下，轻松替换为与自身实现方法完全不同的软件。

怎么做　创建一个用于输入输出文本的命令

设计一个能在命令行使用的读写文本流的软件。

和类似于串行通信协议的接口相比，文本流的简单接口更值得选择，因为这样能方便我们组合使用多个软件来完成不同的工作。

当然，在某些情况下，复杂的二进制数据格式是无法避免的。不过，这种情况并不常见。上述做法虽然会给扫描带来一定的负担，但与自由读写通用的文本流所带来的便利性相比，这点牺牲还是值得的。



UNIX 思想④

分离原则

英 语 Rule of Separation

是什么 从机制中分离出策略

分离原则指从机制中分离出策略。

- **策略**

  策略是依赖于软件前提的部分，在代码中相对不稳定。业务逻辑和用户接口就属于策略。

- **机制**

  机制是不依赖于软件前提的自我独立的部分，在代码中相对稳定。绘图处理的光栅操作算法等可以起到引擎作用的部分就属于机制。

为什么 机制稳定，策略不稳定

将策略与机制结合在一起有两个缺点：一是策略固化，不利于迎合用户需求进行更改；二是修改策略时要被迫修改已经稳定下来的机制。

将策略与机制分离可以消除上述缺点，同时获得以下好处。

- 在尝试新策略时不必担心机制会被破坏
- 便于为机制编写有效的测试代码。机制的寿命比较长，所以编写测试代码是比较合适的
- 独立的机制可以被其他软件重复使用

比如 UNIX 的 GUI（Graphical User Interface，图形用户接口）系统 X Window System 就只实现了机制，没有实现策略。这么做是为了让 X Window System 成为通用的绘图引擎。

X Window System 将 GUI 的风格全权交给其他层次的部件（工具箱等）来处理。这个系统只为我们提供能构筑任何策略（GUI）的机制（绘图引擎）。

之所以这么做，是因为策略的变化远快于机制。用户接口风格的流行趋势日新月异，但光栅操作、图形合成逻辑等并不会经常发生改变。

怎么做 改善分离出来的策略

对软件中稳定的部分和不稳定的部分分开进行管理，比如下面这两种设计。

- **服务类应用**

  将模块分割成前端与后端。负责理解通信协议、接收客户端请求的前端是策略，实际提供服务的后端是机制。

  这样一来，两个模块的复杂度就会比实现相同功能的单一模块低很多，从而降低故障发生的概率，减少软件整体的成本。

- **编辑类应用**

  让用于扩展编辑器功能的模块可通过设置文件来驱动。面向用户的可扩展接口为策略，编辑器的引擎为机制。

  这样做不仅可以明确内部的模块分割，还可以让用户自由地扩展功能。



UNIX 思想⑤

简单原则

英 语 Rule of Simplicity

是什么 代码要简单

简单原则指将代码设计得足够简单。

先探讨是否具有让所有代码都保持简单的条件，然后规定只在必要的情况下允许添加复杂代码。

不过，编程总要面对代码复杂化的压力。要克服这一点，需要整个开发团队认可简单代码的价值。

为什么 代码会自然而然地变得复杂

下面几点会导致代码趋于复杂。

● **程序员的自我表现欲**

程序员是一群能够随心所欲控制代码复杂度和操作抽象逻辑的聪明人。因此，程序员喜欢与同事明里暗里竞争“谁写出来的模块更美”。

但是，由此产生的复杂设计往往超出团队的编程能力或调试能力，结果适得其反，最终导致失败。

● **外部对功能的需求**

决定功能的往往不是用户真正的需求，也不是软件实际能提供的服务。实际上，功能是被市场上多变的主流趋势所支配的。

很多时候，销售方只看中功能的噱头，根本不管该功能有没有人使用。这种来自销售方的压力往往会破坏一些优秀的设计。如今功能方面的竞争已经陷入了“功能越多越好”的怪圈。其结果就是代码大幅增加，产生很多规模巨大且容易出现故障的复杂代码。最终，所有人都深受其害。

怎么做　营造“简单即美丽”的文化氛围

要在程序员之间营造“简单即美丽”的文化氛围。我们可以尝试以下几种做法。

- 坚决抵制代码膨胀及代码复杂化。对简单的解决方法给予好评
- 拒绝用大量功能装饰软件。相反，要探索分割方法，将聚集了大量功能的软件分割成相互协作的多个零件



UNIX 思想⑥

3.43 简约原则

英 语 Rule of Parsimony

是什么 不写大代码

简约原则指不写大代码。这里的“大”既指代码量大，又指代码内部的复杂指数大。

只要不是别无他法，就不要写大代码。

写代码要吝啬，尽量少写，让代码维持“小”的状态。

为什么 大代码无法控制

一旦放任代码变大，其复杂度就会上升，给维护带来障碍。

不可否认，曾经花费大量劳力写出来的代码让人难以舍弃。但随着新代码的加入，代码规模会越来越大，早晚有一天程序会因过于复杂而崩溃。

怎么做 不额外添加代码

首先，编写规模小、复杂度低的代码。

如果代码会随着新代码的添加而变大，我们就得尽快对代码进行分割。除非分割失败，否则坚决不能保留大代码。



UNIX 思想⑦

透明性原则

英 语 Rule of Transparency

是什么 让软件运行变得可见

在设计软件时，要保证我们能从外部清楚看到软件的运行状态。也就是说，我们不仅要让软件正确运行，还要让人能看到它在正确地运行。

为此，在设计时我们要注意下面所讲的透明性和公示性。

- 透明性：软件的运行要让人一眼就能理解“软件正在做什么”以及“软件是怎么做的”
- 公示性：可以监视或表现软件的内部状态

为什么 有助于调试

具有透明性和公示性的设计虽然不会对用户有什么直接的贡献，但对整个项目有着潜在的正面影响。

之所以这么说，是因为当软件的运行状态变得可见时，调试和故障排查会变得更加省力。调试会占用开发周期的大部分时间，在早期引入简化调试的机制是一项收效颇丰的投资。为调试而生的功能绝对不是附属品。

另外，以透明性和公示性为目标，更容易获得用于软件间协作的简单接口，从而便于开发者使用其他开发工具（特别是 Test harness、Profiler 和调试脚本等）操作软件。

怎么做　编写“软件运行可见化”功能

我们可以在软件设计初期加入用于调试的功能，比如加入以字符串形式输出模块内变量内容的方法、大量生成日志等。

在编写一般功能时，将这些能让软件运行可见化的功能一并写入代码中。在正式代码中加入简化调试的机制是一件好事，不必犹豫。



UNIX 思想⑧

3.45 健壮性原则

英　语　Rule of Robustness

是什么　使软件具有健壮性

健壮性原则指让软件具有健壮性。

健壮的软件不仅能在一般条件下正常运行，还能在预想之外的条件下提供适当的服务。健壮性也可以称为坚固性。

不过，编程时如果将预想之外的条件也纳入考虑范围，代码就会变得复杂。随着复杂度的提升，代码最终会超出人脑的理解范围。如此一来，代码发生故障的概率就会大幅提升。软件失去健壮性，陷入进退两难的境地。

所以，要想使软件具有健壮性，必须保证人们能轻松识别其内部结构。为此我们需要编写具有透明性和简单性的代码。

- 透明性：可读性高，软件运行一目了然，这样的代码就是透明的
- 简单性：代码要完成的任务不复杂，所有的分支条件都能简单说明，这样的代码就是简单的

为什么　软件必须足够坚固

与健壮的软件相比，不具备健壮性的软件为用户提供的价值明显偏低。如果一款软件用不了多长时间就出问题，那它肯定无法为用户提供优质的服务。如果一有输入错误软件就会卡死，那么对于这样的软件，用户也不可能买账。

为保证健壮性，软件必须能让人轻松识别其内部结构。如果读代码的人无法轻松说明代码的内部结构，就证明代码不具备透明性，自然也不具备健壮性。另外，在出现问题时，过高的复杂度会妨碍维护人员理解代码，使维护人员无法通过修改代码来保证软件的健壮性。

怎么做 让代码透明化和简单化

要时常保持代码透明和简单。为此，我们需要做到以下两点。

- **代码审查**

  如果编写代码的程序员无法准确说明代码的内部结构，就表示这段代码暗藏危险。

- **检查软件是否能承受例外输入的考验**

  将其他软件给出的结果用作输入可以提升测试效率。相较于手动输入，这么做更能对自己编写的软件形成压力。



UNIX 思想⑨

表达性原则

英　语 Rule of Representation

是什么　尽量用数据表达信息

代码中的信息要尽量用数据而非逻辑来表达。

将信息固定在数据一侧，可以提高逻辑的可读性与健壮性。

数据不论多么复杂都能被轻松地模型化。用数据来表达信息也更加简单。

为什么　数据比逻辑更好控制

对人类而言，过程逻辑并不是很好理解，但复杂的数据结构理解起来并没有那么难。所以说数据比逻辑更好控制。

比如用于表示换算表的数组的初始化语句与表示相同内容的 `switch` 语句相比，数据结构一方更容易理解。不论表达能力还是信息量，数据结构都有着明显的优势。

怎么做　把复杂的部分交给数据

我们要尽量把代码中无法避免的复杂部分交给数据表示。当遇到数据和逻辑必须有一个复杂化的情况时，我们要毫不犹豫地选择让数据复杂化。

在优化代码时，也可以考虑将代码的复杂性转移给数据。



UNIX 思想⑩

最小意外原则

英 语 Rule of Least Surprise

是什么 接口的设计要符合使用者的想象

接口的设计尽量不要让使用者感到意外。

设计接口时要避开无意义的新奇设计以及过度取巧的设计。

为什么 降低学习成本

什么样的软件用起来最简单？答案是需要用户学习的东西最少的软件，即用户利用已有知识就能进行操作的软件。

这种软件能够按用户的预期运行，没有门槛，不会给用户带来压力，因此能被用户长期使用。

怎么做 活用用户的已有知识

在设计接口时，我们要让接口的运行符合用户的预期。为此，我们需要注意以下几点。

- **参考类似的软件来设计接口**

  我们可以参考用户常用的、功能类似的软件来设计接口。以开发计算型软件为例，就是“+”必须和计算器一样永远表示加法。

- **考虑目标用户的特征**

  最终用户、其他程序员、系统管理者……用户类型不同，能给用户带来最小意外的接口也不同。

- **注意传统**

  在 UNIX 的世界中，配置、运行控制文件的格式和命令行开关等都包含了一些约定俗成的东西。

  传统的形成是为了让学习更加简单。我们要学习这些传统，并将其活用到需求的设定中。

- **避免“相似但稍有不同”**

  人们往往希望眼前熟悉的东西就是自己熟知的东西，但希望越大，落空时的失望就会越大。与其编写一款与现有软件相似的产品，不如直接做一款完全不同的产品。

不光是软件的接口，编程的接口也是如此。假设有一个从模块取值的函数 `get`。如果要获取远程模块的值，就不应该使用“get”这个名称。虽然从取值的意义上来说，这个名称没有问题，但“get”会给人一种获取当前模块中某个值的感觉。当获取远程模块的值时，准备一个完全不同的 `fetch` 函数可以降低使用者的意外程度。



UNIX 思想⑪

沉默原则

英　语　Rule of Silence

是什么　软件应“沉默寡言”

软件应将显示内容控制在最少的程度，默默执行自己的工作。

除非遇到必须提醒的情况，否则软件应“什么都不说”。

不论对软件的用户而言，还是对和软件相连接的软件而言，“沉默寡言”的软件都要比“喋喋不休”的软件好相处。

为什么　能更好地传达重要信息

“沉默是金”同样适用于软件。

如果输出内容过多，用户就难以分辨哪些信息对自己来说比较重要，还有可能在下拉滚动条时错过重要的信息。

如果软件能保证只显示重要的信息，用户就不必自己再去筛选信息了。用户的注意力和精力是宝贵的有限资源，好钢要用在刀刃上。

另外，软件的输出有时会用作其他软件的输入。此时的信息如果玉石混淆，就会影响到软件与其他软件的连接，所以软件必须保证能让其他软件轻松提取所需信息。

有些软件在运行开始和运行结束时会分别显示“应用程序正在启动”和“应用程序正在关闭”的信息。其实这些信息并没有什么意义，因为它们都是用户知晓的内容，而且与该软件相连接的软件也不会将这些信息用作输入数据。

怎么做　只输出重要的信息

将输出的信息控制到最少。只输出重要的信息，内部运行的信息不要掺杂其中。

下面两条可视为沉默原则的指导方针。

- 在发生错误时只将真正错误的部分作为标准错误输出，不输出所需范围之外的任何数据
- 如果因为调试而需要输出信息来显示当前运行状况，则可以制作一个具有冗余模式的开关，并让开关默认处于关闭状态



UNIX 思想⑫

修复原则

英 语 Rule of Repair

是什么 修复失败时停止处理

在软件运行的过程中，一旦出现错误且修复失败，就应立刻停止处理。

此外，软件发生错误时要有明显的提示。

不论在正常情况下还是在出现问题的情况下，软件的运行过程都应该保持透明。在陷入混沌的局面之前，软件应引发明显的错误，中断处理。

为什么 出错时继续运行容易扩大损害

在遇到预想之外的情况时，软件若能恰当应对，就不会出现任何问题。即便不能，在某些情况下，软件也可以通过修复来回到正常状态。

然而，如果软件在修复失败的情况下继续执行处理，就会出现最坏的结果。故障可能会在不知不觉间破坏所有数据。我们发现问题时已经晚了。

如果只是软件暂时无法正常运行，倒也不算可怕，可怕的是有时候故障会悄悄地破坏用户的重要数据。这种破坏会带来巨大的损失，酿成不可挽回的后果。

怎么做 错误提示要“震耳欲聋”

软件在发生错误时应尽早发出能让人立刻知晓的通知。软件在无法自行修复错误时要停止处理，尽早提醒用户，让用户进行判断，这一点非常重要。

不过，对于用户的非法输入和软件自身的执行错误，软件要先适当地进行一下处理，如果条件不允许，再立刻引发一个容易诊断的错误。

关联信息 软件输入输出相关的箴言

关于软件的输入输出，有一种观点是“接收输入时采取自由主义的态度，发送输出时采取保守主义的态度”。具体来说，就是在接收输入时，即便输入格式有误，软件也应当尽量推测出其正确的含义，宽容接纳；在发送输出时，软件应保证输出数据严谨、整洁和正确。这句箴言原本用于网络软件，但它的思想也适用于其他领域。

不过，对输入过于宽容也会带来一些问题。HTML 就是一个对输入过于宽容从而酿成苦果的典型案例。不同的浏览器会选择不同的超集，这就导致不同的浏览器呈现出的效果各不相同。

软件应当对“运行方式”而不是“运行方式的解释”采取宽容的态度。我们在制定规则时，对于那些“怎样实现都可以”的宽泛标准一定不能妥协。



UNIX 思想⑬

3.50 经济原则

英 语 Rule of Economy

是什么 珍惜程序员的时间

程序员的时间是宝贵资源，值得珍惜。
下面列出的几个问题会浪费程序员的时间。

- **硬件性能不足**

  开发的机器性能不足是浪费程序员时间的常见因素之一。
  当存储空间不足时，压缩和转储操作就会变多，从而消耗大量时间；当 CPU 的性能不足时，编译和执行方面就会花费大量时间，导致程序员的等待时间增加；当内存不足时，可同时执行的软件数量就会变少，导致开发效率低下；当显示器太小时，程序员就要不断切换窗口，这就降低了工作效率。

- **可用软件的限制**

  在无法购买所需软件的情况下，程序员就需要花费很多精力来实现相应的功能。
  另外，有些团队还限制免费软件的安装。在无法安装工具的情况下，很多工作就需要手动完成了。

- **环境相关的规则或限制**

  在开发过程中需要用到某个网站，开发环境却无法访问该网站。这种情况很常见。有些团队还会规定不允许使用云端软件，甚至不允许连接网络。

然而，信息处理技术人员是不能与信息相隔绝的。

为什么　程序员的时间很宝贵

不选择购买硬件或软件通常是为了节约经费。然而设备的费用与花在程序员身上的费用相比，后者要远远高于前者。如果对设备进行投资能提高程序员的工作效率，让程序员舒心工作，那么这部分投资很快就能回本。

另外，保证安全性是制定规章制度的主要目的。但如果规章制度阻碍了开发，降低了商业效率，那就本末倒置了。越优秀的程序员越容易被规章制度束缚，失去工作热情。

怎么做　对设备进行投资

购买一些能优化开发环境的软件和硬件。对设备进行投资就相当于间接在程序员身上投资，如此一来，收支必然为正。这么做可以提高开发效率，减轻程序员的工作压力，从而大幅提高生产效率和产品质量。

另外，实施规章制度后要适度对其进行调整。将最初制定的规章制度视为基础方案，在实施这些规章制度之后，也要听取程序员的意见。规章制度如果阻碍了开发，就要在可能的范围内做出最大的让步。



UNIX 思想⑭

生成原则

英 语 Rule of Generation

是什么 编写“用于生成代码”的代码

生成原则指编写用于生成代码的代码，也就是用我们写出来的代码来生成代码。

我们要减少手动操作，编写用于生成代码的代码。

不必强求所有代码都通过这种方式生成。只要在可能的范围内生成部分合适的代码即可。

为什么 生成出来的代码成本低且质量高

人类并不擅长做细致的工作。因此，手动操作代码会麻痹人的感觉，导致工作进度过慢或工作出现失误。

而生成的代码通常比程序员手写的代码成本低且质量高。这也是编译器和解释器存在的意义。

这一点的正确性大家都知道，但很少有人在日常工作中注意到它。

怎么做 编写代码生成器

我们可以在适当的地方编写代码生成器。

只在有限的范围内生成代码也是很有效果的。

重复的、形式固定的代码尤其适合由代码生成器生成。



UNIX 思想⑮

优化原则

英 语 Rule of Optimization

是什么 代码的正确性优于运行速度

编程中的优化指性能调优，其中包括提高运行速度、优化内存和硬盘等计算机资源的使用率等。

但是在优化代码之前，我们首先要保证的是代码能正常运行。

在代码尚不能充分运行的阶段，对细节进行打磨是一种吃力不讨好的做法。为了那仅有的一点成果，我们往往需要耗费大量的时间。

为什么 在较早的阶段追求运行速度会破坏设计

在没有确认代码能正确运行之前就开始优化会破坏代码的设计。

之所以这么说，是因为此时追求优化会引起以下问题。

- **会牺牲代码的透明性和简单性**

  此时追求优化会使代码变得晦涩，内部构造难以辨认，从而产生诸多故障，浪费大量时间。我们花费大量时间进行调试换来的只是价值极低、效果甚微的高速化，以及一点点计算机资源使用量的减少。

- **贸然对局部进行优化可能会妨碍代码整体的优化**

  对代码整体进行优化能带来最大收益，而贸然对局部进行优化会妨碍代码整体的优化。贸然对局部进行优化不仅会降低软件性能，还会留下许多过于复杂的代码。

怎么做 先确保代码的正确性再想办法提高运行速度

先确保软件能正确运行，然后想办法提升运行速度。在编程时，一定要牢牢记住这个顺序。

未经优化的代码运行缓慢，还会消耗大量的计算机资源。在优化之前，我们要先编写这种未经优化的代码。这一阶段至少要保证代码足够简单，没有过高的复杂度。之后，我们再系统地寻找能获得最大优化效果的地方进行优化。

代码越简单，可优化之处就越容易被找到。

扩展 优化试制品

从优化的角度来说，先做一个能运行的试制品也是一种不错的做法。

通过试制品，我们能发现哪些功能没有必要编写。少写一部分代码自然能提升性能，因为不用写的那部分代码根本谈不上优化。

“最有力的优化工具是删除键”“生产效率最高的日子就是删除了1000行代码的日子”，这些颇具讽刺意味的格言充分说明了代码简洁对优化的重要性。



UNIX 思想⑯

多样性原则

英 语 Rule of Diversity

是什么 容许有多样的选择

软件开发应容许存在多种方法。

软件开发不存在“唯一正确的方法”。我们不能相信任何宣称软件开发存在唯一正确方法的言论。

为什么 人的想象力是有限的

从原理上讲，软件开发不可能存在唯一正确的方法。

人的想象力是有限的。即便有人能让软件的各个地方都实现最优，他也预测不到软件的所有用途。

“存在唯一正确的方法”这种态度既狂妄又不成熟。

怎么做 不懈追求更好的方案

在编程领域，对于任何宣扬“存在唯一正确方法”的言论都应持怀疑态度。在此基础上，我们要认可多样性，调动思维，不断寻找更好的方案。不能认为软件功能存在唯一正确的规格。

另外，只进行一次发布不可能满足用户的所有需求。所以我们要让软件足够开放，并且具有较高的可扩展性，以此来提高软件间相互协作的可能性。我们也可以通过添加选项和 hook 来让用户自定义软件。总之，我们要设计出灵活度较高的软件。



UNIX 思想⑰

3.54 可扩展性原则

英 语 Rule of Extensibility

是什么 设计时留有扩展的余地

我们在设计软件时要考虑到可扩展性。

软件开发不存在“唯一正确的方法”。将自己的设计看作“唯一正确的方法”是很危险的。因此，代码一定要留有扩展的余地。通过扩展，代码能满足多样的需求。

未来比我们想象中来得更快，所以设计一定要着眼于未来。

为什么 软件必须扩展

如果不事先考虑可扩展性，我们就无法在保持软件兼容性的情况下改写代码。一旦如此，软件就会被最初的设计所束缚，无法成长。

最初的设计不可能完美无缺，况且用户的需求还会不断发生变化。

软件若想长期被用户使用，就得不断成长。

怎么做 可插拔式设计

我们要采用可插拔式设计。

对于以扩展为目的的可插拔接口部分，我们要采用灵活的设计，并在代码一旁加上“如果需要 × ×”的注释。这是对将来使用、维护代码的人应尽的义务。即便将来维护这部分代码的人是自己，我们也可能会忘记相关内容。所以这种着眼于未来的设计也为自己提供了方便。

不过，可扩展性并不是添加非必要功能的免罪符。事先添加的功能大多没有用武之地，我们要编写能在需要的时候轻松添加相应功能的代码。

扩展　具有自我描述性的数据格式

可扩展性原则同样适用于数据格式（数据布局与数据格式）。

在设计协议或文件格式时，要让数据格式具有自我描述性和可扩展性。

我们可以通过添加版本号来实现上述内容。不过，让数据格式由几个独立的自我描述的部分组成能取得更好的效果。这么做可以在不搞混所读取的代码的前提下添加新的部分或删除旧的部分。

让数据格式具有自我描述性是一项很小的投资，但这项投资可以使我们获得丰厚的回报——在不影响已有代码的前提下扩展数据格式。



3.55 UNIX哲学

英 语 UNIX Philosophy

是什么 支撑 UNIX 的哲学

UNIX 哲学指 UNIX 背后的设计哲学，也就是 UNIX 思维。

UNIX 哲学可以总结为以下几个定理。

- 小就是美
- 工作唯一
- 尽早创建原型
- 可移植性优先于效率
- 文本数据
- 充分利用软件的杠杆效应
- 活用shell脚本
- 避开交互式用户接口
- 过滤器化

为什么 UNIX 哲学具有普遍价值

UNIX 拥有很长的历史，至今仍被人们用在开发的第一线。这是因为 UNIX 的设计哲学非常优秀且具有普遍价值。

最好的证据就是 UNIX 中使用的点子在其他软件的开发中得到了应用。

怎么做　活用 UNIX 哲学

我们要将 UNIX 哲学活用到设计方针及编程中。

掌握设计哲学背后潜藏的理由能帮助我们选择合适的方针。

接下来的几节笔者将逐一说明 UNIX 哲学的各个定理。

出处

[1] Mike Gancarz. The Unix Philosophy[M]. Oxford: Butterworth-Heinemann, 1994.



UNIX 哲学①

小就是美

英 语 Small is beautiful

是什么 软件规模越小越美丽

软件规模越小越美丽。这里的美丽指“价值高”。

规模小的软件简单、易用，远远优于规模大的软件。

因此，我们在开发软件时应将软件规模控制在较小的范围。维护软件时亦是如此。

为什么 规模较小的软件比较好用

规模较小的软件有以下优点。

- **易于理解**

  规模较小的软件专注于处理一项工作，所以代码相对简单。这种软件只包含最低限度的算法，所有代码都为其工作目标服务。

- **容易维护**

  软件规模小，代码的可读性就比较高。代码的可读性高了，代码维护就会变得简单，因为理解代码是维护的第一步。

  充分理解代码虽然是维护的大前提，但在实际工作中人们往往忽视这一点。

- **给计算机资源带来的负担较小**

  在软件规模较小的情况下，运行只会占用很少的内存。于是，内存分配变得简单，交换与分页的工作也相应减少，性能因此而得到提升。

  另外，硬盘占用量也会变少。

  用一个词总结就是“轻量”。

- **便于同其他软件组合**

  软件体积小，其工作内容和接口也一定不会复杂。这方便软件与其他软件自由组合，从而灵活应对需求的变化。

规模较大的软件存在以下问题。

- **代码复杂，令人难以理解**

  代码规模越大就越难控制。规模巨大的代码超越了人类的理解能力，就连编写代码的程序员有时也会忘记一些内容，比如哪个函数放在了哪个模块、哪个地方相互引用了变量、某个变量的目的是什么，等等。

  这样的代码很难进行调试。

- **无法应对例外情况**

  规模较大的软件自身就是一个独立的世界。这种软件通常会备齐用户需要的所有功能。

  然而谁都无法预测未来，需求是会发生变化的。单独一款软件不可能应对所有的例外情况。

怎么做　不论开发还是维护，软件都要保持较小的规模

在写代码时，从简单的地方开始写起，让软件保持在一个较小的规模。

在设计软件时，要控制软件的规模，让软件专注于一项工作。功能不足的问题可以通过与其他软件协作来解决。

要想让软件保持在较小的规模，就要充分理解需要解决的问题，因为在没有完全理解问题的时候，我们常会想一些复杂的解决方案，使软件规模变大。体积巨大的软件中充斥着与原本目的无关的代码。



UNIX 哲学②

3.57 工作唯一

英 语 Make each program do one thing well

是什么 一个软件只负责一项工作

每个软件只负责做好一项工作。

在自己的生命周期内专注于做好一项工作的软件才是真正的好软件。这种软件在完成自己的工作后会立刻退场，为负责下一项工作的软件让路。

假设有一个用于显示目录内容的命令。我们不能为了美化显示结果而给这个命令添加调整输出格式的功能。这个命令只负责显示目录内容。至于美化显示结果，我们应该另外创建一个命令来完成，然后将两个命令组合起来使用。

为什么 让软件变得纯粹

软件只负责做好一项工作，这样可以消除软件自身多余的代码。多余的代码会拖慢软件的运行速度，还会提升代码的复杂度，对添加功能等后期维护工作造成影响。

软件只负责做好一项工作还能帮助我们抓住该工作的本质。如果写不出这种只做好一项工作的软件，就证明我们对问题的理解还不够透彻。

只负责做好一项工作的软件便于在其他软件中重复使用。我们往往对重复使用大规模软件采取谨慎的态度，而对于这种只做一项工作的软件，我们就不需要有所顾虑了。

怎么做　专注于一项工作

我们要让一个软件只负责一项工作。

在处理规模较大的问题时，先将大问题分割成小问题，然后开发解决各个小问题的小软件。要让这些小软件只专注于一项工作。这样一来，原本规模较大的问题也能一步一步得到解决。

当某一问题又延伸出其他问题时，我们要再开发一个专注于另一项工作的软件，然后组合使用该软件与现有软件，以此来解决整个问题。不在原软件上直接添加功能是为了避免出现面条式代码。面条式代码常在规模巨大的软件中出现。

编写出专注于一项工作的代码后，要时刻提醒自己不去添加功能和选项。抵御“多功能主义”的诱惑比我们想象的要难。一旦经不住诱惑，我们的软件就会一点一点偏离原本的目标。因此，我们要时常问自己，这些代码是否真的有用，是否偏离了软件的本质。



UNIX 哲学③

3.58 尽早创建原型

英 语 Build a prototype as soon as possible

是什么 尽早着手创建原型

要尽早创建出软件的原型。

要想知道一个点子是否可行，是否能变成看得见摸得着的产品，最快的方法就是亲手试着做一下。这个试着做出来的东西就是原型。

等待完整的功能说明书只会浪费时间。通过原型进行学习还能加大开发的动力。

为什么 不经历失败就做不出好软件

没有人能避免失败，没有人能每次都得到正确的结果。软件开发也一样，我们不可能一上来就编写出完美无缺的软件。对软件而言，持续的改良必不可少。我们只有经过不断摸索，对软件进行无数次修改，才能做出一款优秀的软件。

另外，在未创建原型的阶段，所有点子都逃不出“这里应该这样运作”这种臆测的圈子。大部分人无法在这一阶段理解设计构想，结果就出现了一个人一种理解的现象。但是，项目要想向前推进，成员的理解必须一致。这时，原型作为一个看得见摸得着的标志物，有助于使所有成员的想法实现统一。

怎么做　借助原型提高可靠度

我们要尽早创建原型，缩短发布周期。

原型创建得越早，产品的发布也就越早。创建原型能获得以下好处。

- **提早发现前提性质的错误**

  项目初期需要确定设计方针，我们必须尽早判断方针的正误。创建原型能帮助我们提早发现前提性质的错误，将损失控制到最小，从而避免到发布时才发现问题。

- **减少需求不完备导致的返工**

  原型带来的反馈可以减少需求不完备导致的返工。用户见到原型后如果给出积极的反应，我们就可以确信需求没有错误。

  有时候用户会指出不满的地方，这些意见是我们应该加以重视的宝贵信息。与其在开发完成后被大量用户诟病，不如在正式开发之前接受少数用户的批评，未雨绸缪。

- **能提早开始排查错误**

  根据原型带来的反馈，我们可以提早开始排查错误。这项工作开始得越早，我们就能越早完成高质量的最终产品。随着开发的不断推进，有问题的算法、不一致的时间点和难懂的用户接口等问题会陆续暴露出来。如果存在原型，我们就可以提早对这些问题进行排查，在不断摸索的过程中提高产品的质量。

关联信息　第三系统

任何系统都有一个改良的过程，都会按照“第一系统”“第二系统”“第三系统”的顺序发布。

- 第一系统：性能良好，但欠缺一部分必要的功能

- 第二系统：天平倒向另一边，在牺牲性能的前提下添加了很多功能
- 第三系统：性能与功能之间取得了恰当的平衡。该系统保留了真正有用的功能，我们能够用适量的资源完成尽可能多的工作

我们在开发系统时应以第三系统为目标。

那么，怎样才能开发出第三系统呢？

答案是先做出前两个系统。除此之外别无他法。再怎么努力跳过前面的步骤，到头来也只是多做了几个第一系统和第二系统，还不如按照顺序每种系统只做一个。

虽然不能跳过第一系统和第二系统，但我们可以缩短第一系统到第三系统的周期。具体方法就是使用原型。原型是一种手段，它的目的是实现第三系统。



UNIX 哲学④

可移植性优先于效率

英 语 Choose portability over efficiency

是什么 可移植性优先于效率

软件设计由一系列选择组成。其中让人为难的是可移植性和开发效率这一对互斥选项。这里我们应该优先选择可移植性。

可移植性的评价标准是软件在配合其他平台进行修改时所需成本的多少。

重视可移植性有助于软件在诸多平台上运行，延长软件的使用寿命。

为什么 维持软件的价值

衡量软件是否成功的标准之一是软件能在多少种平台上运行。

当软件依赖于特定的硬件时，软件就只能在该硬件有竞争力的时期保持价值。一旦硬件失去竞争力，软件的价值就会下降。我们要想维持软件价值，就需要将软件移植到其他硬件上。这一阶段如果耗时过长，就会给业务带来致命的打击。

开发时将重点放在提升可移植性上的软件能以较少的成本移植到新硬件上。移植所花时间也比较短，因此我们可以用节省下来的时间开发新功能。

为了提升可移植性，我们可以考虑适当降低开发效率，多投入一些劳力。可移植性带来的回报将十分丰厚。

怎么做 编写不依赖于硬件的代码

在设计软件时要重视可移植性。

为了提高可移植性，我们在设计软件时要将依赖于硬件的部分和不依赖于硬件的部分分离。不依赖于硬件的部分应按照便于重复使用的单位实现模块化。

为了提高可移植性，代码优化（性能调优）方面所花时间应尽量缩短。因为一旦深入优化，就需要用到硬件的特殊能力，这显然会对软件的可移植性造成影响。可移植性优先于优化。只要软件拥有良好的可移植性，就算暂时损失一些运行速度，我们也可以等更加先进的硬件问世后，使用新的硬件来解决速度问题。



UNIX 哲学⑤

3.60 文本数据

英 语 Store numerical data in flat ASCII files

中 文 将数值数据保存在ASCII平面文件内

是什么 文本文件优于二进制文件

把数据保存在文本文件中，不要采用二进制格式的数据文件。

数据是在移动、交换和被诸多软件使用的过程中不断提高价值的。因此，数据也要具备良好的可移植性。使数据具备可移植性的方法就是将数据保存在文本文件中。

不论对用户而言还是对程序员而言，在大多数情况下，文本文件要优于二进制文件。

为什么 文本文件是万能的

文本文件有以下优势。

- 文本是最普遍的、可移植性最强的格式。文本格式的数据文件比代码轻便，因此它的可移植性比代码要强
- 方便人们随时查看数据。另外，文本文件能够通过普通的文本编辑器阅览，便于修改，也便于在开发中调试
- 文本格式便于工具和命令调用，不需要像二进制格式那样先转换成其他格式

文本文件确实比二进制文件冗长，但在大多数情况下，二者运行效率的差距小到可以忽略不计。

怎么做　标准文本文件

我们要把数据保存在文本文件里。

设计的软件要可以输入输出文本文件，要便于和其他软件进行协作。

文本文件应选用不受语言和操作系统限制的标准格式（CSV、XML等）。这样可以提高软件与其他软件的连接性。



UNIX 哲学⑥

3.61 充分利用软件的杠杆效应

英 语 Use software leverage to your advantage

是什么 通过软件的杠杆效应增大力

杠杆是由硬棒和支点组成的一种可以增大力的装置。

在杠杆的一端（施力点）施加力后，另一端（作用点）会产生反方向的力。当支点位于硬棒中央时，施加的力与作用力的关系为 1 比 1。将支点向作用点移动后，作用点产生的力将大于我们施加的力，这就是杠杆效应。

将软件用作杠杆，就可以使自己的力增大多倍。要想实现杠杆效应，就需要知道如何让支点接近作用点。放到软件领域来说，就是想办法重复使用既有软件，将各个软件组合起来使用。

个体的力是有限的。不过，将多个个体的力聚集起来，然后加上杠杆效应，力就能增大很多倍。

为什么 用较少的劳力获得巨大的成果

一次性编写大量代码的最好方法是借用现成的代码。

所谓借用代码，就是把其他软件的模块、设置文件或软件本身拿到自己的软件中使用。有句格言是这么说的：“好的程序员能写出好的代码，伟大的程序员能借来好的代码。”

将前人的努力成果融入自己的工作之中，能够大幅提高自身代码的可用性。

作为被使用的一方，这些现有软件通过在更多的软件中发光发热，

可以使自身存在的价值得到提升。而使用现有软件的软件，因为投资减少，所以能获得更多的收入。

总而言之，我们可以将杠杆效应应用到软件工作中。

怎么做　将手动作业自动化

不要编写充斥着各种功能的大型软件，而要编写多个功能单一、价值单一的小型软件。然后通过胶水语言将这些小型软件连接起来，共同完成一项大任务。

将手动作业自动化是有效利用软件杠杆效应的一个典型示例。对于某些工作，我们有时会不自觉地手动完成。但计算机能完成的工作由人来做就是浪费时间。我们要养成将手动作业自动化的习惯，提升操作的正确性和生产效率。



UNIX 哲学⑦

活用shell脚本

英 语 Use shell scripts to increase leverage and portability

中 文 利用shell脚本加大杠杆效应，提高软件的可移植性

是什么 通过 shell 脚本进行连接

我们可以利用 shell 脚本加大杠杆效应，提高软件的可移植性。

- **杠杆效应**

  为了最大限度利用软件的杠杆效应，我们需要有效使用 shell 脚本。也就是说，用 shell 脚本将软件与其他软件或命令连接起来。

- **可移植性**

  为了保证软件的杠杆效应，我们要使用 shell 脚本提高可移植性。shell 脚本属于解释型语言，不需要编译成平台专用的二进制文件，所以与编译型语言相比，shell 脚本在提高软件可移植性方面效果更好。

为什么 加大杠杆效应

shell 脚本能发挥巨大的杠杆效应。

shell 脚本中使用的命令不需要我们亲自编写，直接使用他人的成果即可。我们可以将这些命令添加到 shell 脚本中，从而获得杠杆效应，实现自己的目的。

另外，shell 脚本有着很高的可移植性。能轻易从一个平台移植到另一个平台的软件往往会被更多的人使用。软件的用户越多，其杠杆效应就越明显。

怎么做　将 shell 脚本用作胶水语言

将 shell 脚本用作胶水语言。用 shell 脚本将小型软件连接起来，共同完成一项大工程。

我们要抵挡住诱惑，不把自己熟悉的编译型语言用作胶水语言。使用编译型语言会影响软件的可移植性，编译过程还会消耗额外的时间，打乱编程的节奏。况且，shell 脚本与编译型语言在执行速度上本就相差无几。

关联信息　胶水语言

胶水语言的英文是 glue language。glue 是胶水的意思，胶水语言就是连接软件与软件的语言。

最常用的胶水语言是 shell 脚本。

shell 脚本又称为命令行解释器，它有简单的控制结构。利用 shell 脚本可以使命令与命令、软件与软件相结合，实现流程作业，或者完成特定的处理。



UNIX 哲学⑧

3.63 避开交互式用户接口

英 语 Avoid captive user interfaces

是什么 避开交互式用户接口

交互式用户接口又称为捆绑型用户接口，是软件位于命令解释器的上位与用户进行对话的一种模式。

这种软件从启动到运行结束，用户都无法与命令解释器对话。用户被捆绑在该软件的用户接口内部，在解除捆绑之前无法脱身。

为什么 交互式用户接口会束缚用户、机器及软件

交互式用户接口会产生以下问题。

- **程序员需要记住软件各自的对话方法**

  在 UNIX 中，程序员原本只要记住命令解释器的使用方法即可。但如果使用了交互式用户接口，程序员还必须记住软件各自的对话方法。

- **软件之间无法对话**

  用户接口以人为基准进行了优化，shell 脚本失去了杠杆效应。

- **等待时间变长**

  等待人类的输入成了机器的瓶颈，机器无法充分发挥性能。

- **输入部分负责解析的代码变得膨胀和不清晰**

  如果将输入的多样性和输出的可读性纳入考虑范围，解析输入字符串的代码必然会发生膨胀。

- **产生“大就是美”的错误引导**

  交互式用户接口用菜单进行功能选择，这正是多功能主义蔓延的契机。此外，交互式用户接口无法与其他软件协作，所以只能让自己变得越来越大。

怎么做　将控制权还给命令解释器

在设计软件时要避开交互式用户接口。

和软件进行对话的应该是软件，而不是人。软件完成一项工作后，应当立刻将控制权还给命令解释器。

当然，为初学者准备的交互式用户接口在很多情况下有着很好的效果。但随着用户的不断学习，交互式用户接口会变得冗长无用，所以我们要准备两种用户接口，一种面向初学者，一种面向老用户。



UNIX 哲学⑨

3.64 过滤器化

英 语 Make every program a filter

中 文 让所有软件都成为过滤器

是什么 把软件设计成过滤器

我们要把所有软件都设计成过滤器。

所谓过滤器，就是将输入流作为数据接收后，经过某种加工，再将数据以流的形式输出。

软件的本质是处理数据，而不是生成数据。为了最大限度发挥软件的能力，所有软件都应该设计成过滤器。

为什么 软件的意义就是输入输出

软件是人写出来的，所有软件都是处理数据的过滤器。

比如实时收集数据的软件就是过滤器。软件要定期取样，收集数据，这就是输入流。然后，软件会从数据中抽取合适的部分，以流的形式输出。

再比如，拥有 GUI 的软件也是过滤器。GUI 将鼠标和键盘按键视为事件来处理。这些事件会发送给窗口系统控制下的某个软件，这就是输入流。软件对事件有所反应，更新显示器的显示内容，这就是输出流。

计算机是高效收集、过滤数据的道具。我们设计的软件只要能很好地完成这两项工作即可。

怎么做　使用标准输入输出

要将软件设计成处理数据的过滤器。

特别是在设计可以通过命令行启动的软件时，一定要正确设计成使用标准输入输出。数据输入要使用标准输入，数据输出要使用标准输出。另外，错误信息要用标准错误输出。

这种设计方式可以保证软件之间能够进行连接（可连接性）。这样一来，软件之间就能相互组合，共同完成一项由单独的软件无法完成的大型工作了。

扩展　UNIX 哲学中的准则

UNIX 哲学中有十条值得我们重视的准则。

● **允许用户自定义环境**

让用户可以按照自己的喜好调整环境。

UNIX 的用户喜欢根据自己的想法修改环境。因此，许多 UNIX 的软件在使用风格的选择上给了用户非常大的自由度。

用户最初会因 UNIX 的灵活性而感到迷茫，被大量的选项吓到。但时间一长，用户就会花时间去学习这些选项，并试着使用它们。当适应到一定程度后，用户便很难再用回其他操作系统了。当花费在 UNIX 上的时间累积到一个阈值时，用户就会改掉系统中不顺眼的部分。

这种情况并非只出现在操作系统中。当软件具有足够的灵活性时，用户会为了更好地使用软件而花费时间去学习使用方法。而且，用户越习惯自己调整的环境，就越难回到那些无法调整的环境中。

● **让内核小而轻巧**

操作系统的内核要小而轻巧。

在 UNIX 的历史中，关于操作系统的内核中要不要包含应用程序这一问题，人们讨论过很多次。因为在内核中加入应用程序可以在应用程序的执行过程中减少上下文切换的次数，提高性能。

然而，UNIX 拒绝了内核肥大化，选择了让内核小而轻巧的方案。原因有以下两点。

一是如果应用程序包含在内核里，普通程序员就无法对其进行修改。只能由负责内核的程序员进行维护的应用程序无法实现进化。

另一个原因是当应用程序出现故障时，内核很可能受到影响。在最坏的情况下，使用该应用程序会导致意外停机。

● **使用小写字母**

命名时要使用小写字母，而且名称要短。

使用小写字母是 UNIX 的传统。命令名和文件名基本都使用小写字母。另外，名称要尽量短小简洁。

名称使用小写字母的原因有以下几点。

首先是对眼睛好。小写字母更适合我们长时间阅读。

其次，小写字母的外观富于变化，所以小写字母有较高的识别度，更易于阅读。

最后，能使大写字母更加突出。如果周围的文件名都是小写字母，这时我们将一个文件命名为“README”，这个文件就会非常醒目。另外，在排序时，大写字母会排在小写字母的前面，这也让大写字母变得更加突出。

使用较短的名称是为了将命令控制在一行以内。特别是用管道操作符连接命令时，将命令缩短到一行会带来许多方便。

● **保护森林**

尽量不用纸，保护森林。

UNIX 用户讨厌纸质文档，因为他们无法操作纸上的数据。印刷在纸上的数据既不能排序，又不能移动，也不能筛选、变形和修改。至少这一切不能像在计算机上一样轻松实现。纸上的数据不

能保存在硬盘里供我们随时取出使用，也不能进行搜索，当然也不能通过加密来保护重要信息。

我们要养成在计算机中保存文本文件的习惯，并学会使用强大的工具来操作它们。

- **沉默是金**

软件应极力减少显示内容。

UNIX 的命令在需要显示详细错误信息的情况下也会保持沉默。用惯了其他操作系统的用户可能会感到不适应，可一旦习惯了 UNIX，就会深谙其好处。

以用于显示文件一览的命令为例，当目录中不存在文件时，其他操作系统会显示“没有找到文件”之类的报告，而 UNIX 不会显示任何信息。

软件保持沉默，画面上就只会显示有意义的数据，不会出现无意义的信息。这就能防止有用的信息被淹没在无用数据的大海之中。

另外，保持沉默还有利于软件之间相互结合。UNIX 命令发挥的是过滤器的作用，它通过 UNIX 的管道结构与其他过滤器相结合。很多看上去对用户很友好的信息，往往会成为阻碍软件间相互结合的主要因素。

- **并行思考**

所谓并行思考就是让 CPU 尽可能忙碌。CPU 的速度非常快。为了最大限度发挥 CPU 的能力，我们允许 CPU 与周边装置或人类的步调不一致，让 CPU 时刻处于忙碌的状态。

为此，我们要把工作分割成多个部分同时执行。大部分工作是可以分割成多个部分的。几个部分同时执行能提高工作效率。

- **部件联动**

部分的总和大于整体。

将小部件聚集起来组成一个大规模的软件要比单独开发一个大规模软件更有价值。

小部件组合而成的软件具有很好的灵活性。加入其他部件可以延长软件的使用寿命。

而拥有多个功能的综合型软件属于大规模单体型软件。这种软件包含多余的功能，而且无法只针对一部分功能进行扩展。另外，庞大的代码量使这类软件难以维护，执行时需要使用较多的资源也对性能造成了不好的影响。

● **完成 90% 的目标**

我们要以完成 90% 的目标为准则。

任何事情都很难达到 100% 完美。保证做好 90% 的工作能大幅提升工作效率，性价比很高。要以满足对象用户 90% 的要求为目标，剩下的 10% 保持“用户自己想办法解决”的状态。

所谓完成 90% 的目标，就是故意无视最难的部分。最难的部分是所有问题中最消耗成本的部分，也是最花时间、最难编程的部分。忽视最难的 10%，剩下的问题将很快得到解决。

● **劣就是优**

UNIX 重视劣等功能的生存能力。该操作系统体现了“劣就是优”这一逆向思维。

我们时常能听到“UNIX 不如其他操作系统”这样的评价。UNIX 受到过很多批评，比如“用户接口难用”“太简单，不像一个正经的操作系统”等。然而，如果 UNIX 真有这么多地方劣于其他系统，为什么那些系统没能经得起历史的考验生存下来呢？

UNIX 相信“最大公约数”类型的系统能生存下来。“虽不高级但有效率”比“高质量且高价”的系统更容易让人们接受。

● **层次化思维**

结构要有层次。

UNIX 用户和 UNIX 程序员喜欢将事物整理成层次化结构。UNIX 文件系统使用的目录结构是最早的层次化结构。UNIX 的其他地方也用到了这种层次化思维，比如进程树、X 窗口系统和网络服务等，都取得了成功。

自然界的秩序是符合层次化结构的。UNIX 的结构以自然为模型，证明了层次化思维是一种非常优秀的思维方式。

层次化思维本身确实十分简单。UNIX 的所有方面都体现了这一思维方式，因为这个简单的思维方式中蕴含着巨大的意义。




第 4 章　视角

~ 程序员的视角 ~

身上缺点太多固然是一件坏事，
但是满身缺点却又不愿意承认，
那就更不好了。

——帕斯卡






内聚度

英 语 Cohesion

别 名 模块强度

是什么 模块要“纯粹”

内聚度用于表示模块内功能的纯粹程度，用来衡量模块的强度。

内聚度的强度分为七个等级。等级越高的模块越纯粹，强度也越高，质量也越好。

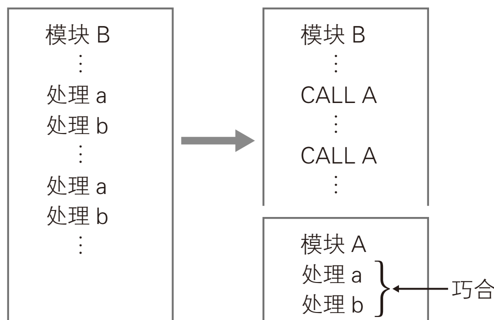
判断内聚度要着眼于模块内各元素之间关系的紧密程度。下面笔者按照等级由低到高（由劣到优）的顺序逐一进行说明。

① 等级一　巧合强度

在巧合强度模块中，各元素之间并不存在特别的关系。“巧合”指事物恰巧一致。

比如模块内恰好有重复的命令群模式，我们将它另外整合成一个模块，这个模块就是巧合强度模块。在这种情况下，我们无法准确命名、定义该模块的功能，因为模块处理的功能各不相关。

我们可结合下图来理解巧合强度。



模块 A 中的处理 a 和处理 b 没有什么特殊的关系。只是因为这种命令群模式恰巧在模块 B 的多个地方出现，所以我们将其整合成了一个模块。

模块 A 中包含了很多没有任何关联的功能。对该模块而言，模块的部分命令发生变更的可能性还是很高的。这时，如果模块 B 要调用模块 A，而模块 B 的其余部分（相同部分之外的部分）又与变更的内容毫无关联，那么调用模块 A 将是一种很危险的行为。如此一来，整个模块设计都要重新修改。

另外，在巧合强度模块中，各个元素与该模块内其他元素的关联性较弱，这就表示这些元素很可能与其他模块中的元素具有较强的关联性。因此，当其他模块发生变更时，巧合强度模块就容易受到影响，变得难以维护。

再者，无法准确定义模块的功能就意味着模块的功能很难被重复使用。

② 等级二　逻辑强度

逻辑强度模块指抽象地整合了某种功能的模块。

所谓抽象地整合某种功能，举例来说，就是把所有的输入输出操作整合成一个模块，或者把编辑各种数据的功能整合成一个模块。

这种模块内的逻辑会根据条件来选择不同的执行路径。也就是说，逻辑强度模块包含了几种相互关联的功能，调用模块会识别其中一种来执行。

所以，这种模块被调用时只会执行其中一部分命令，不会执行模块内的所有命令。模块内部命令群的关联性会变弱，模块强度会变小。

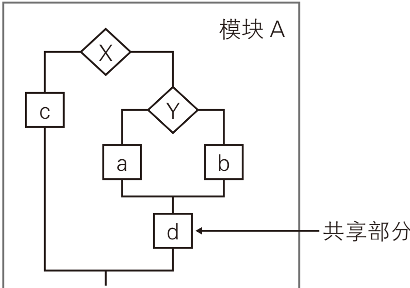
另外，不同的功能只能使用同一个接口（输入输出参数）与调用方模块联系。因此，参数的操作方面容易出现编程错误。

不过，逻辑强度模块也有好的一面。

首先，这种模块包含多个共同的功能。部分逻辑可以实现共享。

其次，相关功能整合在了一个模块内，这有助于让程序员的思维更加集中。

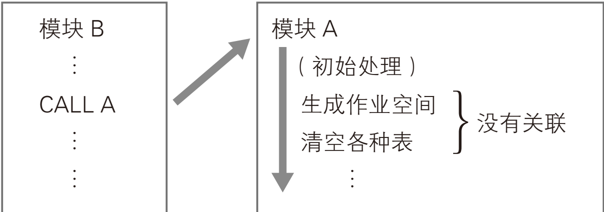
再次，特定数据的处理能够集中在一个模块内进行。也就是说，逻辑强度模块有着良好的信息隐蔽性。这样一来，当该部分数据发生变更时，我们只要对这一个模块进行修改即可。



③ 等级三　时间强度

时间强度模块由在特定的时间点连续执行的多个功能整合而成。模块内各功能之间不存在很强的关联性，它们只是在特定的时间点被连续执行而已。

时间强度模块中比较有代表性的是“初始处理模块”。在软件的初始处理中，生成作业空间、清空各种表等处理虽然不存在很强的关联性，但它们都会在软件的初始处理中统一完成。



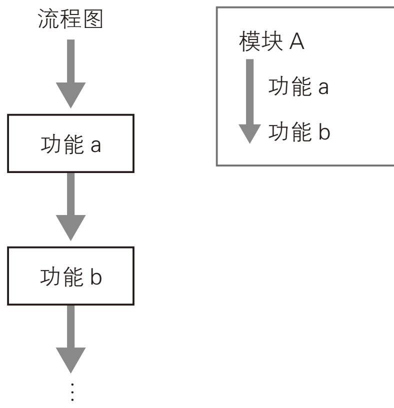
④ 等级四　流程强度

流程强度模块由处理问题时所需的全部或部分功能组合而成。

这些功能按顺序执行。一般来说，流程强度模块具有时间强度模块的特性，但由于流程强度模块的各功能之间具有流程方面的关联性，所以这类模块的内聚度比时间强度模块的内聚度要高。

流程强度模块在我们只需用到一种功能时无法使用。要想只使用这类模块中的一种功能，就必须让该模块实现功能选择。但这样一来，模块的内聚度就会降至逻辑强度。

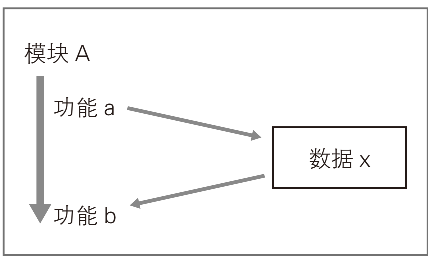
如果将大型功能的一部分流程整合成一个模块，那么这个模块就是流程强度模块。把流程图的部分内容模块化就属于这种情况。



⑤ 等级五　通信强度

通信强度模块基本拥有流程强度模块的特性。

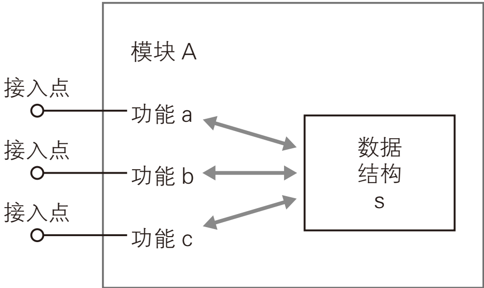
与流程强度的模块不同的是，这种模块内部的各功能之间会进行数据的传递（通信）或引用相同的数据。在通信强度模块中，因为功能在数据上存在联系，所以通信强度模块的内聚度要高于流程强度模块的内聚度。



⑥ 等级六　信息强度

信息强度模块由处理特定数据结构的多个功能组成。

这种模块的基本思想是同一种数据结构（信息）尽量由同一个模块来访问。这样一来，修改数据所带来的影响将限制在相应的模块内，从信息隐藏的角度来说有一定的好处。



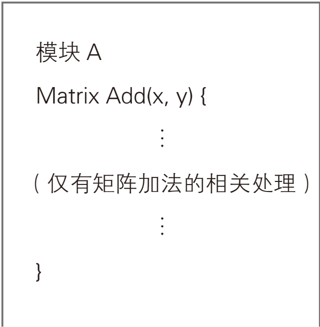
逻辑强度模块也具有信息隐藏的特质，只不过在接入点的数量方面与信息强度模块有所不同。

逻辑强度模块只有一个接入点，执行的功能要靠参数来选择。

而信息强度模块有多个接入点，各接入点执行单一的固定的功能。各个接入点拥有自己的参数，因此这类模块不存在逻辑强度模块中参数难以处理的缺点。

⑦ 等级七　功能强度

在功能强度模块内，所有命令都为了完成一项工作（功能）而相互关联。这类模块是纯粹度最高的模块。



为了实现一个功能，所有命令都相互关联。

在这种情况下，仅修改一部分命令的可能性非常小。即便要进行修改，也是因为使用该模块的其他模块都出现了相同的问题。

也就是说，功能强度模块可以将修改限制在该模块内部。这类模块对其他模块的影响程度比其他强度的模块要小。

为什么 有杂质的模块很脆弱

代码要分割到模块中进行管理。

独立性对分割好的模块而言十分重要。提高模块的独立性有助于减少代码的复杂度，增加代码的可靠性，对代码的维护也有帮助。

要提高独立性，就得让各个模块内元素的关联性达到最强，同时让模块与模块之间的关联性减到最弱。

内聚度是衡量模块内各元素关联性的尺度，是评价模块自身关联性强弱的标准。

内聚度低的模块有一些典型的特征，比如模块会处理一些不相关的工作，或者模块需要处理的工作过多等。这会导致以下问题发生。

- 代码难以理解
- 代码难以维护
- 代码难以重复使用
- 代码脆弱，容易受修改的影响

内聚度高的模块，其内部各元素间有很强的关联性。这种被适当细分化的模块会专注于执行特定的工作。这能带来以下好处。

- 使代码的设计更加明确，更容易让人理解
- 代码易于维护和扩展
- 促进代码的重复使用
- 促进代码之间的低耦合性

怎么做 以实现高内聚的模块为目标

为了提高模块的独立性，我们要尽量实现功能强度模块。

不过，从可以将特定数据的处理控制在局部范围这一点来看，信息强度模块也有很高的实用价值。虽然信息强度和功能强度是模块非常理想的状态，但模块并不是非要达到这种状态不可。在实际进行模块化时，由于要兼顾各种情况，我们有时也需要实现其他强度的模块。

即便迫不得已需要实现某种强度的模块，我们也不能省掉讨论代替方案的过程，否则会给整体设计带来不好的影响。

出处

[1] 国友義久. 効果的プログラム開発技法 [M]. 東京：近代科学社，2009.

相关图书

[1] Glenford J. Myers. Composite/structured design[M]. New York: Van Nostrand Reinhold, 1978.



耦合度

英 语 Coupling

是什么 模块间应“疏远”

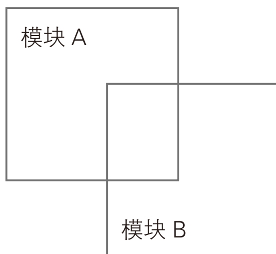
耦合度是衡量模块之间关系紧密程度的标准。耦合度测量的是两个模块之间耦合的松紧程度。

耦合度分为六个等级。等级越高（数字越大），关联程度就越弱，耦合度就越低，模块质量也就越好。

耦合度的判定要着眼于模块间如何传递数据。下面笔者会按照等级由低到高（由劣到优）的顺序逐一进行说明。

① 等级一　内容耦合

内容耦合指模块与其他模块存在共享部分。



直接引用其他模块内未进行外部声明的部分，或者与其他模块共享一部分命令的情况都属于内容耦合。

内容耦合不会出现在使用高级语言编写的模块中，但常在使用汇编语言编写的模块里出现。

在内容耦合的模式下，一个模块的修改会对其他模块造成影响。修改模块时兼顾其他模块是一项非常费力的工作，所以在内容耦合的模式下，修改操作很可能会引发故障。

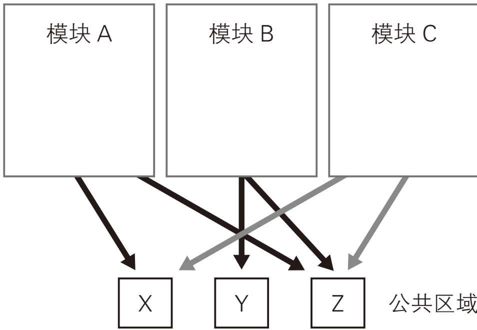
② 等级二　公共耦合

公共耦合指在公共区域内定义的数据由多个模块共同使用。

公共区域内定义的数据就是全局变量。

公共耦合的耦合度很高，害处有很多。

- 公共区域的数据不会出现在模块间的接口上，这使得代码不容易被解读
- 因为公共区域的数据可以供毫无关系的模块使用，所以代码的安全性较低
- 公共耦合的模块通过公共区域的数据与许多模块发生关联，阻碍了代码的重复使用



当数据 X 的长度因模块 A 发生改变时，负责模块 A 的程序员可能不知道模块 C 也在使用数据 X。负责模块 A 的程序员可能没有想过修改数据 X 会给模块 C 带来什么样的影响。这样一来，模块 C 很可能出现意想不到的问题。

不过，公共耦合的数据可以被其他模块使用。这一点既是公共耦合的短处，也是它的长处。

如果避开公共耦合，模块之间传递数据的参数必然会增多。传递数

据的参数一旦增多，我们就很难正确理解每一个参数的意义了。创建模块时也需要花费更多的时间来写参数。

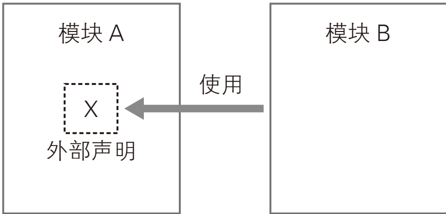
如果使用公共耦合，我们就不用指定参数了。这可以降低创建模块的难度，其带来的优势甚至能弥补前面所说的短处。现实中确实有不少采用公共耦合的事例。

然而，采用了不恰当的模块设计才是需要大量参数传递数据的主要原因。通过重新设计模块，再次审查数据的位置，参数的数量大多能够减少。

③ 等级三　外部耦合

外部耦合指模块间共享外部声明的数据。

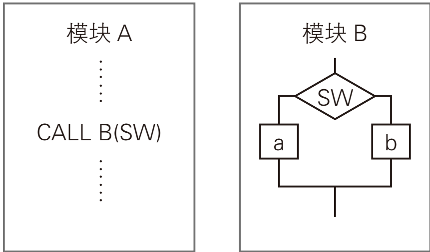
外部声明的定义，举例来说就是 `public` 声明的函数。



从共享数据的意义上讲，外部耦合与公共耦合有相似之处。不过，外部耦合只外部声明有用的数据，所以不会像公共耦合那样连没有用的数据都共享。从这一点来说，外部耦合的耦合度要低于公共耦合的耦合度。

④ 等级四　控制耦合

控制耦合指调用方模块以参数的形式传递涉及被调用方模块内部控制的数据。

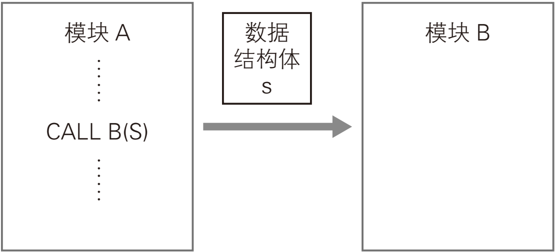


在控制耦合的模式下，调用方模块以参数的形式将选择变量传递给被调用方模块，从而指示被调用方执行相应的功能。因此，调用方必须知道被调用方模块的逻辑，不能将对方当作黑箱。这就增强了耦合度。

控制耦合还有一个缺点，那就是被调用方模块的内聚度会达到逻辑强度。不过，因为不会共享数据，所以控制耦合的耦合度要低于公共耦合和外部耦合的耦合度。

⑤ 等级五　特征耦合

特征耦合指两个模块间传递公共区域中没有的数据结构。



数据结构通过参数传递。

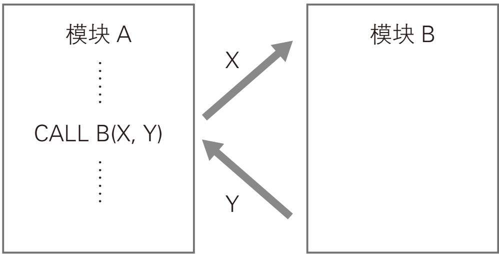
在特征耦合的模式下，传递的数据结构并不是都能用到的。这一点提高了特征耦合的耦合度。

⑥ 等级六　数据耦合

数据耦合指模块间仅通过参数传递标量类型的数据作为模块间的接口。

因为可以将对象模块视为黑箱，所以数据耦合的耦合度最低。

在模块间的耦合中，通过明确的参数传递数据的数据耦合最为优秀。



通过上图可以看出，模块 A 将数据 X 传递给模块 B，用作模块 B

的输入数据，然后接收模块 B 输出的数据 Y。至于模块 B 是如何将数据 X 转换为数据 Y 的，模块 A 不需要知道。也就是说，模块 A 可以将模块 B 视为黑箱，模块 B 的逻辑修改不会对模块 A 造成影响。

模块 B 接收数据 X，输出数据 Y，从内聚度来看属于功能强度，同样很优秀。

值得一提的是，特征耦合传递的是数据结构，并且只使用其中一部分数据结构。如果接收数据的一方会处理数据结构中的所有数据，那么这种耦合也可以视为数据耦合。

为什么　相互依赖的模块很脆弱

耦合度高的模块会相互依赖并且相互影响。很多问题便由此产生。

比如修改相关模块之后，所有与其耦合的模块也要进行修改，模块的运行也会受到影响。

另外，在使用某个模块时，必须同时使用与其耦合的模块这一点对模块的重复使用造成了影响。

再者，高耦合度模块的代码很难单独拿出来理解。即便结合相关模块的代码一起阅读，全局变量的存在也会让阅读代码的人难以掌握该模块整体的运作方式。

怎么做　以实现低耦合的模块为目标

为了提高模块的独立性，我们要尽量降低模块间接口的耦合度。

具体做法如下所示。

- 尽量通过参数传递数据
- 数据尽量不用全局变量保存。只用一次的数据存放在局部变量里
- 不让模块的运行因外界传来的值而发生改变。如果参数传来的标志位内容是“A”就执行添加，是“D”就执行删除，那么模块的耦合度就会变高

我们还要注意与当前模块相耦合的模块的质量。虽说高耦合度本身就是一个问题，但究其本质，其实是模块与不稳定要素的耦合在作祟。模块依赖于稳定的库就不会带来什么问题。

与其盲目追求数据耦合，不如根据耦合对象来判断模块间耦合到哪种级别比较合适。

扩展一 混合耦合

有观点认为，耦合度的分类中还包括混合耦合。

混合耦合表示数据在不同状态下可以存在多重意义。比如计算税率的函数在计算不合法时会返回负数值来表示出错。

- 正常时：税率
- 异常时：负数值

在这种情况下，返回值会根据具体情况而发生变化。这就要求函数使用方在使用函数时必须注意到这一点。因此，混合耦合的耦合度略高。

扩展二 耦合的个数与方向

对于耦合度，我们要把着眼点放在耦合的“强度”和“松紧程度”上。

而对于耦合，我们要注意的地方就不止这两个了。除了要看“强度”和“松紧程度”，还要注意“个数”与“方向”。

与其他模块有双向关系的模块，拥有和高耦合度模块一样的弊端。

关联信息 幂等性与安全性

有观点认为，模块与函数具有幂等性和安全性。

- **幂等性**

  幂等性是数学术语，表示对于某操作，无论执行多少次都会得到相

同的结果。同一个函数无论被调用多少次，得到的结果都是一样的。比如 0 的乘法计算就是幂等的。“3 × 0”“(3 × 0) × 0”“((3 × 0) × 0) × 0”全部等于 0。

另外，绝对值计算也是幂等的。“abs(−3)”“abs(abs(−3))”“abs(abs(abs(−3)))”的结果都是 3。

- **安全性**

  安全性指不让操作对象的状态发生变化。

  而副作用表示使状态发生变化，所以安全性还可以说成“对操作对象的状态没有副作用”。

具有安全性和幂等性的服务器能让客户端放心地以操作黑箱的方式使用服务。对于 HTTP 请求等在易于发生错误的网络环境中使用的协议，这两个特性尤其有用。

比如 HTTP（HyperText Transfer Protocol，超文本传输协议）的 GET 请求就要求是幂等且安全的。这样一来，发送 GET 请求后如果没有返回响应，用户只要重新发送一次请求即可。因为可以保证获取相同的结果，所以即便前一次的请求也被处理了，对服务器来说也不会有什么实质性的影响。

出处

[1] 国友義久. 効果的プログラム開発技法 [M]. 東京：近代科学社，2009.

相关图书

[1] Glenford J.Myers. Composite/structured design[M]. New York: Van Nostrand Reinhold, 1978.

[2] 山本陽平. Web を支える技術—HTTP、URI、HTML、そして REST[M]. 東京：技術評論社，2010.

[3] 伦纳德・理查森，山姆・鲁比. RESTful Web Services 中文版 [M]. W3China 徐涵，李红军，胡伟，译. 北京：电子工业出版社，2008.



4.3 正交性

英 语 Orthogonality

是什么 代码独立

正交在几何学中指两条线段如直角坐标系一般相交且成直角。让某点沿 x 轴平移时，x 值会发生变化，但 y 值不变。也就是说，x 值的改变不会对 y 值造成影响。

代码也应满足正交性。也就是说，代码之间应具有独立性和分离性。

假设有两组代码。如果修改其中一方不会对另外一方造成影响，那么这两组代码就是正交的。正交的代码在修改时不容易出现问题。

比如访问数据库的代码和用户接口代码就可以设计成正交的。这样一来，我们就能在不对访问数据库的代码造成影响的情况下修改用户接口，还能在不修改用户接口的情况下更换数据库。

为什么 正交的代码更牢固

让代码具有正交性可以获得以下两点好处。

- **提高生产效率**

  代码具有正交性后，修改会控制在局部范围。开发周期和测试周期会因此而缩短。此外，代码的耦合度也会降低，这有助于代码的重复使用。

- **降低风险**

  代码具有正交性后，发生问题的部分可以被立刻隔离起来，代码也

由此变得更加牢固。另外，代码的依赖性较弱，便于测试和检验。

怎么做 代码层次化

我们要让模块间的耦合度最小化。也就是说，不对其他模块公开不必要的信息，与此同时，保证代码不依赖于其他模块的实现。

层次化是将代码之间的耦合度最小化的有效手段。软件应该由包含一个个独立功能的模块组合而成。这些模块要分层次整理、抽象化。各层次只使用下一层次提供的抽象化功能。这样一来，下层代码的修改就不会对上层代码造成影响，灵活性大大提高。

此外，代码的层次化使模块之间的关系得到整理，降低了模块间依赖程度呈指数上升的可能性。

反过来，使用全局数据会最大化代码间的耦合度。代码一旦使用了全局数据，就会与其他共享该数据的成分紧密关联起来。为了满足正交性，一定不要使用全局数据。

扩展 层次化的好处与坏处

层次化是用于分割复杂软件系统的方法。这一方法既基础又重要，许多领域的架构设计使用了该方法。

在运用层次化的示例中，最具代表性的就是网络协议。我们以网络协议为例来看一下层次化的好处。

- 可以将一个层次视为整体，不必熟悉其他层次。比如即便不熟悉以太网的运作方式，也能理解在TCP（Transmission Control Protocol，传输控制协议）上搭建FTP（File Transfer Protocol，文件传输协议）服务的方法
- 可以在不影响其他层次的前提下置换整层代码。比如以太网、PPP（Point to Point Protocol，点对点协议）及网络运营商的改变并不会影响FTP服务的运行

- 可以最小化各层次间的依赖程度。比如网络运营商变更了与IP（Internet Protocol，网际协议）有关的物理传输系统，在这种情况下FTP服务并不需要进行修改
- 层次结构适合标准化。TCP/IP就是定义了层次运作方法的标准通信协议
- 实现层次化后，许多高水平服务就可以使用层次结构了。TCP/IP被ftp、telnet、ssh和HTTP使用。没有TCP/IP，高水平协议就必须创建独立的低水平协议

不过，层次化也有弱点。

首先，层次之间的修改可能会发生连锁效应。比如通过软件用户接口添加字段的情况。字段要保存在数据库中，因此从用户接口到数据库之间的所有层都要添加该字段。

另外，添加层次后，性能可能会变差。数据每通过一层就要转换一次格式，这种转换会给性能带来负担。

关联信息一　关系的正交性

正交性这个词在不同的语境下有不同的含义。

在关系数据库的基础理论“关系模型”中，有一个与规范化同等重要的概念。这个概念就是“关系的正交性”。这里所说的正交性与代码的正交性意义不同。

正规化指从一个关系（= 表）的内部消除重复。

而关系的正交性是一个与多个关系间的重复有关的概念。从编程的角度来讲，这个概念更接近于 DRY 原则。简单来说就是不含相同的值。

下面我们通过例子来看一下含有相同的值是一个什么样的状态。

- **副本**

  复制表后，值会出现重复。

● **同类型的关系**

按年度分成多个表这种设计就无法保证不出现相同的值。

通过正规化消除了单个关系中的重复后，如果从数据库的整体来看仍存在重复，那么关系之间依旧是相互矛盾的。关系是真命题的集合，某个关系与其他关系不能同时含有相互矛盾的事实。

因此我们需要消除整个数据库的重复，保证关系的正交性。

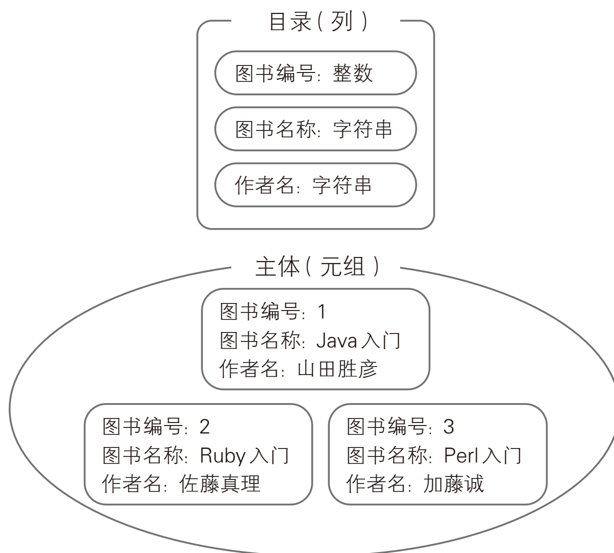
关联信息二　关系的定义

经常有人把关系错误理解成“表与表之间的关系”，认为关系模型用于（通过 ER 图等）设计表与表之间的关系。这其实是一种误解。

实际上，关系模型中的关系是 SQL 中的“表”。

关系模型中的关系由“目录”和“主体”成对构成。目录是多个属性的集合，这些属性由名称与数据类型成对构成。主体则是聚集了属性值的元组集合。

放到 SQL 中来说，关系就是“表”，属性就是“列”，元组就是“行”。



出处

[1] 安德鲁・亨特，大卫・托马斯. 程序员修炼之道 [M]. 周爱民，译. 北京：电子工业出版社，2011.

相关图书

[1] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal. Pattern-oriented software architecture[M]. New York: Wiley, 1996.

[2] 马丁・福勒. 企业应用架构模式 [M]. 王怀民，周斌，译. 北京：机械工业出版社，2010.

[3] 埃里克・埃文斯. 领域驱动设计 [M]. 赵俐，盛海艳，刘霞，译. 北京：人民邮电出版社，2010.

[4] 奥野幹也. 理論から学ぶデータベース実践入門～リレーショナルモデルによる効率的な SQL[M]. 東京：技術評論社，2015.



可逆性

英 语 Reversibility

是什么 选择可以“UNDO”的方案

可逆指发生某种变化后，可通过添加条件回到原本状态的性质。

程序中的判断应时常保持可逆性。禁止使用无法还原的方案。

我们不可能永远都用同一个方案来解决问题。现实世界瞬息万变，受其影响的软件必须由灵活且适应性强的可逆代码构成。

即便方案已经确定，发生问题时也要保证代码能回到之前的状态，这样能将风险控制在局部范围，使风险最小化。

为什么 不存在所谓的最终方案

程序员总倾向于寻求独一无二的最终方案。

然而最终方案并不存在。万事万物都在发生变化。如果过度依赖某一个事实，那么当该事实发生变化时，代码就会出现问题。

比如我们在使用依赖于框架的设计时，会遇到开发到一半被迫放弃使用该框架的情况。理由可能是安全问题、许可证价格问题等，总之有很多外部因素是程序员无法控制的。

一旦发生这种情况，我们就要推翻前面所有的设计。这将带来巨大的损失。

怎么做　不依赖于特定技术

为了能经受住变化的考验，我们的设计需要能回到之前的状态。

为此，代码一定要有灵活性，不依赖于特定技术，能灵活应对变化。

人类不可能每次都做出最正确的决策，所以我们只能让代码随时应对修改。举个例子，我们会遇到在项目中途更换数据库管理系统的情况。如果事先考虑了可逆性，访问数据库的部分也实现了抽象化，那么数据库管理系统的更换操作就能在几个步骤之内完成。

不过，我们不可能准确预测未来。写代码时做到未雨绸缪是一件很难的事。一旦没有把握好度，设计就会变得复杂。

所以我们要考虑风险发生的概率以及风险发生时所造成的影响，在可逆设计与简单设计之间寻找平衡点。

扩展　架构的可逆性

在编写代码时，最重要的是保持设计的灵活性。

此外，在编写详细代码之前，也就是在架构的设计阶段，我们就应该将可逆性纳入思考的范围了。

软件与外部的交界面，如配置平台或第三方组件等就很适合实现可逆性。为了让这些部分具有灵活性，设置间接化的层次等方法必须在架构设计阶段就确定下来。

出处

[1] 安德鲁・亨特，大卫・托马斯. 程序员修炼之道 [M]. 周爱民，译. 北京：电子工业出版社，2011.



代码中的“坏味”

英 语 Bad smell in code

是什么 不要放过代码的凶兆

代码中的坏味指代码中难以理解、难以修改、难以扩展的部分。这些可能潜藏问题的部分是不祥的预兆，是一种特殊的信号，我们把它称为“坏味”。

程序员必须能嗅出这股“坏味”，并对其进行优化。

为什么 嗅觉是重构的必要条件

重构对代码的优化来说必不可少。

重构指在不改变代码外部行为的前提下，对代码内部的结构进行优化。重构也称为“代码体质优化”。

整洁的代码读起来轻松，改起来轻松，调试起来也轻松。而对于不整洁的代码，我们不管做什么都很麻烦。当手边有不整洁的代码时，我们需要在不产生新故障的前提下将其改写为整洁的代码。这种行为或者说技术就是重构。

代码是有生命的。对我们来说，软件发布并不意味着结束，之后我们还要解决故障、添加功能，有时还要将构成软件的模块拿去给其他软件重复使用。如果修改代码时只跟着代码的走向进行修改，添加功能时只顾眼前，代码就会越来越不整洁。这样一来，阅读、修正和调试都会愈加困难。这和不进行健康管理的人身体会变得越来越差是一样的道理。

对人而言，如果多花些时间在看似无用的健康管理上，体质就能有所改善。同样，代码也可以通过看似无用的重构来优化体质，使代码变

得易于修改，功能的添加变得更加安全。

不过，在通过重构优化代码时，我们必须能判断出哪些代码需要优化。我们可以从重构的目录上获取一部分信息。不过，目录上写的情况与我们面对的情况大多不一致。

所以，对于设计上经常出现的问题，我们要了解相关知识，保证能及时发现问题。也就是说，我们要能嗅出这些设计上的“坏味”。

怎么做　了解代码出现“坏味”时的征兆

我们要了解代码出现“坏味”时的征兆。

下面几种情况会导致代码中出现“坏味”。对于每种情况，我们不但要知道什么状态是有“坏味”的，还要了解为什么这种状态会有“坏味”。

- **代码重复**

  代码重复指的是相同的代码散布在软件各处的状态。复制、粘贴代码的行为会导致重复的代码大量存在。

  重复的代码会加大修改的难度，因为在发生故障的情况下，我们要同时修改多个位置的代码。

  我们应将重复的代码整合到一个函数里。

- **函数太长**

  函数太长指的是向下翻几页都看不到函数尽头的状态。

  函数太长会影响理解。在这种情况下，我们无法将函数的功能总结成“这个函数用于实现 A”这样简短的一句话。函数的说明将变成“这个函数是实现 X、Y、Z 和……”。

  我们应将过长的函数分割成若干个小函数。

- **模块太大**

  模块太大指的是模块因规模太大而难以管理的状态。

  模块过大，表明该模块承担的责任过重。过大的模块让人难以理解。另外，当模块的责任过重时，模块被修改的概率也会增加。

随着时间的推移，模块将越来越大，越来越难以管理。

我们应将过大的模块分割成多个小模块。

● **模块太多**

模块太多指的是模块因数量过多而难以管理的状态。

大模块要分解，但不能分解得太小，否则模块数量就会增多，反而难以管理。

减轻责任是有限度的。大模块让人难以理解，模块过多也一样，因为模块增多意味着关联增多，这样一来，我们就难以掌握代码的流程了。

当模块过多时，我们可以删除用处不大的中介模块，或者通过废除、合并等方式减少模块的数量。

● **名称不一致**

名称不一致指的是名称与实际代码不一致的状态。

名称很重要。代码中会出现大量名称。这些名称的作用是向读代码的人传递恰当的信息。在编写代码时，程序员必须注意命名的正确性。

名称并不是一成不变的。代码有生命，随着修改的不断进行，一开始正确的名称到后面也会变得不合适。

当我们发现代码表达的概念与名称不相符时，一定要立刻更正名称。

出处

[1] 結城浩．Java 言語で学ぶリファクタリング入門 [M]. 東京：SB クリエイティブ , 2007.

相关图书

[1] 马丁・福勒. 重构：改善既有代码的设计 [M]. 侯捷，熊节，译. 北京：中国电力出版社，2003.

[2] 乔舒亚・科瑞夫斯盖. 重构与模式 [M]. 杨光，刘基诚，译. 北京：人民邮电出版社，2006.

[3] 罗伯特・ C. 马丁. 代码整洁之道 [M]. 韩磊，译. 北京：人民邮电出版社，2010.



4.6 技术负债

英 语 Technical debt

是什么 问题代码是“欠款”

编程有两条路，一条是花费大把时间写出整洁的代码，另一条是快速写出不整洁的代码。

如果时间充裕，应该坚持选择“花费大把时间写出整洁的代码”这条路。但如果时间不够，修改又比较紧急，有时也可以选择“快速写出不整洁的代码”这条路。

不过，当我们选择“快速写出不整洁的代码”这条路时，软件就等于背上了“债务”。这就是技术负债。

技术负债中的“债务”是代码中难以修改、难以理解的代码。它并不是指故障本身，而是指会给故障创造条件以及阻碍故障排查的问题代码。我们把这个有问题的代码比作“债务”，也就是“欠款”。

在商业中，暂时背负债务的情况并不少见。只要能及时还清欠款，就不会有任何问题。然而债务是会产生利息的。拖久了，利息就会增多，还款也会变得困难，甚至会出现无法还清的情况。

这种情况在代码中也是存在的。

即便暂时写下了不整洁的代码，只要我们能及时还清欠款，就不会有问题。这里的还清欠款是指让代码回滚到修改之前，然后用充裕的时间编写正确的代码，也就是把不整洁的代码修改成整洁的代码。

然而，如果不能及时还清欠款，就会产生利息。不整洁的代码在软件里逗留的时间过久会使问题严重化，这就是“利息”。阅读负债的代码很费时间，而且一次次的修改会让代码变得越来越不整洁，整个软件

的运行也会变得不稳定。于是，利息越滚越多。

等到负债过多的时候，我们就会陷入无力偿还的境地。那时，代码将变得不稳定且无法修改。既不能添加功能，又不能维护。

这样一来，软件便失去了为用户提供价值的能力。

为什么 技术负债会恶性循环

技术负债会恶性循环。

代码一旦变得难以理解，添加功能所花费的时间就会增多。另外，寻找故障、修复故障所需要的时间也会增加。

所需时间增加，就意味着尚未还清的债务会进一步膨胀。这就浪费了一部分时间，使团队无法完成原本应该做的事情。

不过，在软件开发中，“借用时间”是一种常见的策略。该策略可以让我们完成需要完成事情，达成具有一定风险的目标。在有限的时间内我们可能无法编写出完美的代码，但我们可以放低要求，编写出满足基本需求的代码。这样，即使软件暂时处于不完美的状态，我们也可以通过对软件进行管理来弥补“放低要求”带来的损失。

然而，如果欠款一直拖着，我们就会陷入无力偿还的窘境。在没有还清欠款的情况下继续借债，只会让软件陷入更大的危险。

所以我们需要在编程中引入技术负债这个概念，对欠款加以控制。我们不能对欠款采取避讳的态度，而要承认它的存在并与它和睦相处。

怎么做 管理问题代码

我们要在编程中引入技术负债这个概念，管理问题代码。

一想到技术负债的利息，很多人就会觉得代价过高，自己负担不起。然而当我们需要在尽快修复故障和蒙受巨大损失之间进行选择时（在出现严重故障的情况下），快速修复不整洁的代码一定是正确的选择。

不过，解决完故障并不代表万事大吉。技术负债越快偿还越好。用不整洁的代码赶在时限之内成功发布后，应当立刻让代码回滚到原先的

状态，将代码修改成原本就应该实现的整洁的代码。趁自己还记得哪些代码有问题，应尽快还清所有欠款，然后借下一次发布的机会把修改的部分添加到软件中。这种做法才是最保险的。这就好比信用卡，我们买东西时欠下了债务，只要在还款日之前付清，就不必支付利息。

只要能保证如此运用债务，我们就能在快速应对商业需求的同时，避免项目陷入欠款的无底洞。

如果实在没有时间还债，至少要把本应编写的代码设计记录在文档中。

扩展一　技术负债用于说明代码整洁的意义

“技术负债”这个比喻能帮助那些对代码不是很熟悉的人理解代码整洁的意义。“代码能运行就可以，没有那么整洁也无所谓”“没必要去修改能运行的代码”“运行即正确”……持有这类想法的人并不少。只有编写代码和维护代码的人才能切身体会到保持代码整洁的必要性。对于一些不写代码的上司，我们可以把技术负债的概念讲给他听，这么做或许能争取到一些时间。

扩展二　问题代码出现的原因

以下几个要素是诱发问题代码出现及扩散的原因。

- 经验不足的程序员
- 交付日期的压力
- 可读性低的代码
- 特殊化的代码
- 无用的复杂代码
- 低质量的设计

另外，以下这些团队文化也是诱发问题代码出现及扩散的原因。

- 不提倡写整洁的代码
- 接纳不明确的代码
- 不给重构留时间
- 发布之前才做回归测试
- 没有勇气替换包含大量依赖关系的旧系统
- 随意创建分支

我们要对这些诱因加以注意，防止不整洁的代码蔓延。

对于诱因不属于以上内容的问题代码，应将其当作技术负债进行管理。

关联信息一 技术病

在技术负债中，我们将问题代码比作“欠款”，其实这类问题代码也可以用“病”来比喻。

如果在“生病”的软件中修改某部分内容，那么其他没有关联的部分也会遭到损坏。放任不管的话，“病”不仅不会痊愈，反而会逐渐恶化，所以必须实施“手术”。就像外科医生敢于实施手术治疗患者病痛一样，我们要敢于修改代码，积极改良系统。

对系统而言，“手术”就是重构。重构需要花费时间与劳力，但在项目进行的过程中，这部分投资足以收回本钱。不仅如此，还会获得数倍的收益。

再者，治系统的“病”也可以让项目成员获取经验。这些经验能加深团队成员对软件本质的了解，让他们成为这款软件的专家。

因此，我们应积极地进行重构，不能害怕修改。躺在原地一动不动，身体只会越来越差。

需要注意的是，改善操作要慢且精。另外，每执行一次改善操作就要测试一次。一次性进行大幅度的修改是很危险的。一旦修改出现问题，之前所做的改善都会付之东流。

关联信息二 临时解决方案

与技术负债类似的概念还有“临时解决方案”。

在时间紧迫的情况下为渡过难关而采取的措施就是临时解决方案。

临时解决方案应单独处理，不能放在正在开发的软件中。临时解决方案在最开始的时候是当作草案来创建的。由于时间紧迫，在创建时无法完全遵守本应遵守的规则，所以临时解决方案以将来会进行修改为前提。

不过，临时解决方案中潜藏着一种“惯性法则”。这个法则就是临时解决方案一旦出台，就会变为既成事实。临时解决方案已经存在，而且发挥了作用，团队成员接受后，就不会觉得需要去尽快修改它了。实际上，临时解决方案也需要尽早整合到系统中，但重要的工作实在太多，于是这项工作就被搁置了下来。

结果，临时解决方案挂着“临时”的名字，一直放在那里得不到修改。本来是临时抱佛脚的产物，但它却长期留存了下来。

不过，在整个项目中，一个临时解决方案都不创建是不现实的。这会让我们来不及解决某些问题。

所以，对于临时解决方案，我们应采取与技术负债相同的方式进行管理。

出处

[1] James Shore, Shane Warden. The Art of Agile Development[M]. Sebastopol: O’Reilly Media, 2007.

相关图书

[1] 乔舒亚・科瑞夫斯盖. 重构与模式 [M]. 杨光，刘基诚，译. 北京：人民邮电出版社，2006.

[2] 凯佛林・亨尼. 程序员应该知道的 97 件事 [M]. 李军，译. 北京：电子工业出版社，2010.




第 5 章　习惯

~ 程序员的日常 ~

习惯实际上已成为天性的一部分。

——亚里士多德






5.1 程序员的三大美德

英 语 Three great virtues of a programmer

是什么 程序员要懒惰、急躁和傲慢

程序员的三大美德总结出了一个优秀程序员应该具备的特质。

这三个美德分别是懒惰、急躁和傲慢。具体定义如下所示。

- **懒惰**

  懒惰指不遗余力地减少整体劳力的消耗。为了让后来人更加轻松，应编写高效率的代码。

- **急躁**

  急躁指在计算机偷懒的时候会感到愤怒。当计算机的工作效率令人不满，或者计算机没有按照预想的方式工作时，应立即修改代码。另外，编写代码时不仅要注意眼前的问题，还要预想今后可能出现的问题。

- **傲慢**

  傲慢指拥有极强的自尊心。保持这种自尊心，促使自己写出让别人挑不出毛病的代码。

为什么 懒惰、急躁和傲慢可以让工作结构化

这三大美德乍一看都是一些不好的德行，其实并非如此。勤勉、宽容和谦虚更符合这三大美德所表达的意思。

然而对程序员来说，懒惰、急躁和傲慢这种说法更加合理。这么说是有原因的。

- **懒惰**

  懒惰并不是指心态懒散不愿工作。

  在我们的日常生活中，有很多需要重复许多遍的工作。对于这种工作，我们应当采取“懒惰”的态度，尽量不自己动手，而是交由软件自动化完成。也就是说，让计算机来完成这些工作。

  这样一来，那些需要多人重复多遍的工作就可以用较少的时间来完成，而且工作的正确性也能得到保证。

- **急躁**

  急躁并不是指冲着计算机发火。

  计算机有时会不按我们预想的方式工作。这时我们应当急躁起来，动手修改代码，让计算机按照我们预想的方式工作。

  这样一来，用在忍耐计算机上的时间就转变成了顺利进行操作的时间。工作质量将得到提升，完成工作所需要的时间也会缩短。

- **傲慢**

  保持较强的自尊心并不是指对周围的人飞扬跋扈。

  自己写出来的代码，应该不羞于见人。要对代码有责任感。对工作成果应“傲慢”，要以专家的态度对待工作。

  这样一来，我们就可以养成编写整洁代码的习惯了。

  软件离不开维护。如果此时代码的可读性高，易于修改，那么软件维护起来也比较简单，代码的运行速度也会变快。

怎么做　自动化、雏形化、模块化

具体来说，我们应采取以下行动。

- **懒惰**

 让重复的工作结构化。

 对于手动作业，我们应先写代码，再做工具，最后实现自动化。注意，不能一上来就追求自动化。先明确手动作业的流程，再将能自动化的部分自动化，这样做的效率比较高。在进行自动化之前，要先找出自动化效果最好的地方。不管什么时候，只要有重复的工作，就可以考虑使用工具实现自动化。

 另外，结构化也适用于文档。文档创建好之后，我们可以保留它的格式，以便重复使用。这里要注意的是不要去整理文档。保证文档能够被搜索到才是我们需要重视的地方。所以，我们要把搜索文档时可能用到的关键字尽量添加到文档的内容或名称中。相较于在分好类的文件夹中进行搜索，全文搜索的效率更高。

- **急躁**

 我们在工作时要预想哪些问题会发生。提前思考软件的哪些部分可能会出现需求，在客户提出意见之前做好应对的准备。

 此时重要的是分清软件可能被修改的部分与不会被修改的部分，让可能被修改的部分保持足够的灵活性。以不发生改变的部分为基础创建雏形，对会发生变化的部分做变量化处理，保证能自由修改。

 不过，用户是任性的，他们的需求五花八门。全部满足这些需求显然不现实。此时我们可以更急躁一点，通过设置文件来让用户自定义软件。

- **傲慢**

 自己做的工作要能拿得出手，维护要无愧于人。对自己写的软件应抱有专家意识，保证维护能轻松进行。

 此时重要的是要按照功能对写好的软件进行模块化管理。这样，在开发其他软件时，我们就可以重复使用这种按功能整合的模块了。

关联信息　高强度工作对程序员而言没有意义

在职业美德中，经常会出现“拼命”“忘我”等表达高强度工作的

词语。但是对程序员而言，高强度工作是没有意义的。

不可否认，长时间泡在办公室里会让人陷入一种自己为项目做出了很多贡献的错觉。其他人也会有相同的感受。然而事实恰好相反，我们越削减自己在工作上花费的时间和精力，对项目的实际贡献就越大。

编程是一种干到老学到老的工作。随着工作经验的积累，程序员对问题领域的理解会逐步加深，实现相同目标所花费的时间与精力会慢慢减少。只要在工作中注意哪些地方浪费了时间，并将经验运用到下一项工作中去，就能实现工作的效率化。

另外，编程是一种高度的脑力劳动，加班带来的恶劣影响要远远超过我们得到的东西。编写高质量的代码需要集中注意力，拖着疲惫的身躯是写不出好代码的。

再者，我们在身体状态不佳时容易迁怒于人，把所有的过错都推给别人，觉得自己是受害者。这样一来，人际关系就会恶化，团队氛围会变得越来越差。这对团队、成员以及即将发布的成品来说，都是很大的打击。就算这种情况只是暂时的，我们也需要花费一些时间来摆脱这种状态。

自己拼命工作也好，外人施加压力也好，都不能帮助我们提升大脑的工作效率。在脑力劳动方面，工作时间与劳动成果并不成正比。

出处

[1] 小飼弾. 子飼弾の「仕組み」進化論 [M]. 東京：日本実業出版社，2009.

相关图书

[1] 拉里・沃尔，等. perl 语言编程 [M]. 何伟平，译. 北京：中国电力出版社，2002.

[2] 凯佛林・亨尼. 程序员应该知道的 97 件事 [M]. 李军，译. 北京：电子工业出版社，2010.

[3] 斯科特・博克顿. 项目管理之美 [M]. 李桂杰，黄明军，译. 北京：机械工业出版社，2009.



5.2 童子军规则

英　语 Boy Scout Rule

是什么　“打扫”完代码再离开

美国童子军有一项很简单的规则，那就是“离开自己所在的地方时，这个地方必须比来时更干净”。就算弄脏露营场地的不是自己，也必须打扫干净再离开。

这个规则同样适用于编程。在离开之前，要让代码变得比自己来时更整洁。代码最初是谁写的并不重要，重要的是我们要努力改善它。

为什么　抑制代码腐烂

代码必须时刻保持精简。

然而在现实中，代码常常会随着时间的流逝逐渐腐烂，质量越来越差。编写代码时如果太过随意，就会招来这种后果。我们必须阻止这种情况发生。

如果所有人都能遵守“改善完代码再离开”这样一个简单的规则，系统的质量就不会恶化下去，甚至有可能会慢慢提升。该规则还有助于培养程序员的大局观，促使程序员着眼于整个软件的质量。

我们只能通过阅读源代码来掌握软件的行为。代码就是一切。程序员的任务就是不断改善代码。

怎么做　代码要改善之后再提交

对于代码，我们要养成先改善再提交的习惯。这也有助于保持代码简洁。

笔者没有夸大其词。量变会引起质变。

我们的目标不是在提交之前让所有代码都变得完美，而是让代码比之前更优质一点。我们可以对代码进行各种各样的改善，比如调整变量名、分割大函数、去除重复、缩减条件语句、取消循环引用、通过添加接口让使用方法和实现方法分离等。不管改善了什么，改善了多少，都没有关系。

扩展　编程讲究“稳中求胜”

在软件开发中，相比抄近道节约时间与成本，绕远道追求质量能获得更好的结果。童子军规则也说明了这一点。凡是跟代码有关的问题，只要选择抄近道，在维护阶段必然会产生不必要的成本。

比如以下几种情况。

- **在无法直接获取价值的情况下，省去单元测试**

  按照这种方式写出来的软件以后将很难修改。因为在修改的时候，我们无法确定这么改是不是有问题。即便是一处微小的修改，也需要我们手动进行测试。因此，软件变得十分脆弱，维护成本也大幅提高。

  设计本身也是如此。与考虑了测试的设计相比，没有考虑测试的设计因为没有研究使用方法，所以不存在什么优势。

- **为削减成本，强行使用与目的不符的现有系统**

  强行使用与目的不符的系统会让我们在编程的时候感到痛苦。使用这样的系统早晚会出问题。到头来我们还是得按照需求重新设计架构。

此时的成本与正常开发需要的成本相比不但没有减少，反而因为返工而大幅增加了。

- **发现选了不合适的库却不进行处理**

  随着需求的增加，为了隐藏库中不合适的部分，我们需要额外添加层次。而每添加一个层次，分离层次的难度都会进一步增加。这样一来，软件不可能进一步得到优化。

  抄近道的地方一旦成了新增功能的基础，之后的修改操作就会产生巨大的成本。

可见，欲速则不达。

出处

[1] 罗伯特・C. 马丁. 代码整洁之道 [M]. 韩磊，译. 北京：人民邮电出版社，2010.

相关图书

[1] 理查德・蒙森・哈裴尔. 软件架构师应该知道的 97 件事 [M]. 徐定翔，章显洲，译. 北京：电子工业出版社，2010.

[2] 凯佛林・亨尼. 程序员应该知道的 97 件事 [M]. 李军，译. 北京：电子工业出版社，2010.



5.3 性能调优的箴言

英 语 Proverb of performance tuning

是什么 “好”的代码胜过“快”的代码

所谓性能调优，就是编写运行速度快的代码。性能调优也称为代码优化。

很多人认为加快代码的运行速度是一件好事。但实际上，过早优化代码会产生各种问题。

因此，对于代码优化，我们要遵守以下规则。

① 不要在编程之初就对代码进行优化

② 编程之初暂时不要对代码进行优化（适用于专家）

代码优化并不是我们在编程之初就应该考虑的事情。在编程时，我们要注意的是代码的正确性和可读性，编写高质量的代码，而不是想方设法让代码的运行速度变快。

为什么 “快”的代码得不偿失

优化代码需要我们付出无法接受的代价。即便完成优化，代码也会失去一些重要的东西，比如以下几点。

- **可读性变低**

  优化后的代码肯定比优化前的代码更难懂。

因为从性质上来说，优化所做的工作是修改代码中原本简单直接的逻辑。优化代码后，逻辑不再简单明快，变得难以表达意图。也就是说，要提高性能，必须以失去逻辑清楚的设计和降低代码的可读性为代价。

最大限度优化过的代码非常难看，我们很难掌握它的处理过程。

- **质量变差**

代码复杂化会导致代码的可读性下降，从而降低代码的质量。在没有明确描述算法过程的代码中，故障很容易被漏掉。

不论回答的速度有多快，答不出正确答案也枉然。说得讽刺一点，优化在给代码加入难以发现的新缺陷方面算是一种切实有效的方法。

- **复杂度增大**

优化会利用特殊的后门强化模块间的依赖性，提升代码的结合度，让代码能够利用一些平台固有的功能。

用如此取巧的方式编写代码会增加代码的复杂度，同时让代码失去可移植性。

慢慢地，代码将越来越不符合优质代码的条件。

- **阻碍维护**

代码复杂化导致代码的可读性下降，从而提升了维护代码的难度。

首先，问题难以被发现，因为代码优化之后，不自然的描述会增加。这样一来，我们就很难追踪处理的流程了。也就是说，优化后的代码是高风险的危险代码。

再者，优化还会对代码的可扩展性产生不好的影响。优化是在给代码设置更多前提条件的基础上实现的。因此，优化会限制代码的通用性和可扩展性。

● **与环境相冲突**

在大多数情况下，优化只能在特定的环境中发挥作用。在某个特定环境下对代码进行优化后，代码在其他环境中运行的效率可能会变低。

比如我们针对某个特定种类的处理器选用了最合适的数据类型。这种做法就可能会导致软件在其他处理器上执行的速度变慢。

● **工作量增多**

对代码进行优化就等于多加了一项工作。

程序员要做的工作非常多。代码如果能成功运行起来，我们就应该先去处理其他紧急的工作，而不是去对代码进行优化。

优化是一项非常耗时的工作。找到问题出现的原因并对代码进行优化并不是一件容易的事情。一旦弄错优化对象，就会浪费大量宝贵的劳力。

怎么做　先写高质量的代码

我们要先写高质量的代码，然后根据需要进行优化。

高质量的代码是在信息隐藏的原则下写出来的。因为各个决定只会在局部范围产生影响，所以代码的修改不会影响到其他部分。先写高质量的代码再调节性能效率更佳。

况且在大多数情况下，“高质量”与“高性能”并不矛盾。按照上述优先顺序写出来的代码只要满足高质量代码的要求，优化时就不会产生多少新的工作。而且代码质量高，我们在做添加工作时也会轻松一些。

另外，写完高质量的代码之后，如果要进行优化，一定要思考其必要性。优化在很多时候不值得我们花费那么多的时间和成本。是否进行优化，要在与解决故障、添加新功能和发布产品等重要工作相比较之后再决定。

扩展一 软件的性能

从整体来看，除了代码，软件的性能还受到很多因素的影响。比如以下几个因素。

- 执行环境
- 部署的设置或者安装的设置
- 使用的中间件
- 使用的库
- 相互运用的旧系统
- 架构

这样一看，一行一行的代码对软件整体的影响十分渺小。除了各行代码之外，还有很多影响性能的因素。

扩展二 架构的性能

在设计架构时，有时我们也需要把性能考虑进去。

这是因为架构的影响范围很广，后期发现问题很难修改。比如在设计通信协议时就要尽量避免影响性能。

我们应当尽早对这些地方进行性能测试。这样既可以不断完善测试环境，又能尽早发现性能问题，抑制成本上涨。

另外，早期的性能测试需要持续进行下去，因为这么做能使我们清楚地看到修改哪里会严重影响性能。如此一来，当出现性能问题时，我们就不用调查整个架构，只看最近添加的修改即可。

扩展三 性能调优的流程

在实际工作中，很多时候我们会因为软件的特性而需要对代码进行优化。

在对代码进行优化（性能调优）时，我们需要在流程方面遵守几项规则。

- **证明优化的必要性**

  首先要再三确认优化的必要性。有时候用户对某部分性能的需求并没有程序员想的那么高。

- **测量性能，找出瓶颈**

  确认需要优化后，我们要先找出瓶颈所在。

  性能出现问题并不代表所有代码的运行速度都很慢，大多是某个特定部分占用了较长时间。这个占去大部分处理时间的部分称为“热点”。

  我们要全身心地寻找这个热点。

- **优化瓶颈部分的代码**

  发现热点之后要对其进行修改。

- **测量性能，确认优化效果**

  不管是代码优化前还是代码优化后，我们都必须对性能进行测量。

  性能差的部分是无法推测出来的。优化的效果也只能通过测量得知。

- **验证优化后的代码是否存在运行问题**

  优化可能会使代码出现一些新的问题。对代码进行优化后，必须认真检查代码是否存在运行问题。

这里再说一下寻找热点的方法。寻找热点时，应利用分析工具，尽可能仔细且准确地检查代码。

另外，由于优化过程中要多次对代码的性能进行测量，所以为了提高效率，我们最好对这部分工作执行自动化处理。

出处

[1] 松本行弘. 松本行弘的程序世界 [M]. 柳德燕, 李黎明, 夏倩, 张文旭, 译. 北京: 人民邮电出版社, 2011.

相关图书

[1] 约书亚・布洛克. Effective Java 中文版(第 3 版)[M]. 俞黎敏, 译. 北京: 机械工业出版社, 2018.

[2] 有沢誠. クヌース先生のプログラム論 [M]. 東京: 共立出版, 1991.

[3] 皮特・古德利弗. 编程匠艺——编写卓越的代码 [M]. 韩江, 陈玉, 译. 北京: 电子工业出版社, 2008.

[4] 布莱恩・柯尼汉, 罗勃・派克. 程序设计实践 [M]. 北京: 人民邮电出版社, 2016.



无我编程

英 语 Egoless programming

是什么 舍弃自我

在编程的过程中应舍弃自我。

具体来说，就是舍弃“自负”与“自尊”，与其他成员合作。积极地把自己写的代码给别人看，请别人提出意见。

看代码的人和把代码给别人看的人都应抛弃“自己更优秀”的自尊心，双方都应纯粹为创造出更好的东西而交流。写代码时不要炫耀自己的能力，应将注意力放在写出更好的代码上。

为什么 抛弃自我，提升质量

程序员常把自己写的代码当成私有物品，讨厌别人指出代码中的故障以及缺陷。

但是，把代码拿给同事看，虚心接受对方的建议，这种开发方式更有利于提升软件的质量，也有利于营造更好的团队氛围，提升团队成员的编程水平。

怎么做 遵守无我编程的十诫

我们在写代码时要将下述“无我编程的十诫”放在心头。

- 理解并接受自己也会犯错的事实

- 你不同于你的代码
- 人外有人天外有天
- 不要在没有沟通的情况下重写代码
- 对技不如己者要尊敬，并且有耐心
- 世上唯一不变的就是变化
- 真正的权威来自知识而不是地位
- 为信仰而战，但也要坦诚面对失败
- 不能闭门造车
- 宽待他人，严待代码，批评的对象是代码而不是人

扩展 自我的度

无我地进行编程是好事，但自我也是人不可或缺的一种品质。将自我与工作完全分离是一件非常困难的事。而且，自我是完成所有事情的原动力，人只有有了自我才能高效地工作。

单方面扼杀自我，牺牲自己为他人和团队服务，是一种“Win-Lose”（一方赢，一方输）的行为。随性而为并不好，但扼杀自我也不会对团队有任何贡献。

我们要控制好自我的度。

出处

[1] 杰拉尔德・温伯格. 程序开发心理学 [M]. 韩江，陈玉，译. 北京：电子工业出版社，2010.

相关图书

[1] 罗伯特・L. 格拉斯. 软件工程的事实与谬误 [M]. 严亚军，龚波，译. 北京：中国电力出版社，2006.

[2] 史蒂芬・柯维. 高效能人士的七个习惯 [M]. 高新勇，王亦兵，葛雪蕾，译. 北京：中国青年出版社，2013.



5.5 一步一步走

英 语 One by one

是什么 循序渐进

编程时要一次只做一件小事。

一件一件做，一点一点来，就像上台阶一样一步一步走。不要一次性处理多项工作。

完成一个小任务后认真检查，没有问题后再开始下一个任务，如此循环。

为什么 “步步为营”更有效率

一次处理一项工作的工作方式更有效率，最终产品的质量也更好。

之所以这么说，是因为一次性处理多项工作容易让这些工作混成一团，最后哪个都处理不好。就算不会造成大问题，两项工作混在一起也会使代码质量下降，在时间上造成浪费。

一步一步进行编程，最后一步操作撤销起来也会比较容易。

当发生问题需要回滚的时候，如果前面坚持一步一步地完成工作，回滚起来就会很简单。只要把顺序反过来，一步一步走回去即可。

比如我们现在只想删除类 A，但类 A 和类 B 的结果在代码中混成一团。此时在不影响类 B 的情况下回滚类 A 就会非常困难。

一步一步进行编程，工作检查起来也比较简单。

比如在写新的函数时，即使该函数还没有被任何地方引用，我们也能先通过编译来检查该函数是否存在语法错误。另外，我们也可以通过

单元测试检查运行方面是否有问题。这样可以让计算机帮我们做不少工作。

一步一步进行编程，新旧代码的替换也会更安全。

在这种情况下，我们不必一次性推翻所有代码，然后全部重写。不妨在保证运行的状态下添加新代码，当新代码能完全代替旧代码时，再一次性推翻所有旧代码。这个方法虽然听上去很麻烦，但失败的风险要小许多。

为什么要一步一步进行编程呢？除了实用性方面的原因，这与心理因素也有一定关系。

一步一步编程意味着程序员能够掌握和控制代码的状态。这样做能去除不确定因素，让人安心工作。

在有心理压力时，人很难像平时一样做出准确的判断。控制好自己的状态也是写出优质代码的必要条件之一。

怎么做　不一次性处理多项工作

工作要一点一点做。

以重构为例，当我们需要在模块间移动函数时，即便发现函数名不合适，也不要在移动函数的过程中修改函数名。正确的做法是等函数移动完成，确认运行没有问题之后，再着手修改函数名。

在测试驱动开发中，不要一口气写完所有代码后再进行测试。当然，也不能先写完所有的测试代码，再写正式代码。应该先写一点测试代码再写一点正式代码，如此循环。

关联信息一　思考也要一步一步来，一点一点想

聪明的人常给人一种不用思考就能说出答案的感觉。

确实，聪明的人不仅输出速度快，正确率还高，仿佛知道这其中的捷径。其实并非如此。

聪明的人并不是用特殊的方法跳过中间步骤来加快思考速度的。他们也需要一步一步进行思考，只不过这些人会在脑中搭建逻辑步骤，每

一步的思考都快速且准确，所以整体看起来才又快又准。

当不能一下子想出答案时，我们不能怪自己脑子不好。要扎实地按步骤来，一点一点地进行思考。

关联信息二　逻辑思考的秘诀

关于逻辑思考，有几个关键点需要我们了解。

- 想立刻获得答案的态度是不正确的。一眼看不出答案时应当继续思考
- 没有经过深思熟虑就下结论的做法是错误的。发现某个东西可以满足条件时不能想当然，要探讨其他的可能性
- 避免反复思考同一件事
- 直接用脑子思考有些困难，不如边写边思考。边写边思考能产生额外的效果。对于想不明白的地方，有时候写下来一看就明白了
- 直觉对逻辑思考来说也很重要。比如，当我们感觉“创建矩阵有助于整理信息”时，不妨先试一试。不过，直觉只能用在思考的过程中。仅凭直觉来获取答案的行为只能说是瞎猜，这可不是一个好习惯

出处

[1] 結城浩. Java 言語で学ぶリファクタリング入門 [M]. 東京：SB クリエイティブ, 2007.

相关图书

[1] 马丁・福勒. 重构：改善既有代码的设计 [M]. 侯捷，熊节，译. 北京：中国电力出版社，2003.

[2] 肯特・贝克. 测试驱动开发 [M]. 白云鹏，译. 北京：机械工业出版社，2013.

[3] 小野田博一. 史上最強の理論パズル—ポイントを見抜く力を養う 60 問 [M]. 東京：講談社，2003.



5.6 TMTOWTDI

英 语 There's more than one way to do it

中 文 做法不止一个

是什么 工具具备多样性是好事

我们在设计拿给别人用的工具（编程语言、DSL 和 API）时，要想办法让工具具备多样性。

这样一来，虽然工具本身会变得复杂，但不管在什么样的情况下，使用工具的人都不需要写复杂的代码。

为什么 对象多样方法也应多样

现实是复杂的。自然语言之所以复杂，是因为要应对现实。自然语言之所以能使现实简单化，是因为自然语言本身是复杂的。正因为自然语言复杂，它才能描述复杂的现实世界。

软件在本质上也是复杂的。所以，工具作为软件，也应该设计得复杂一些。

虽然方法越少给人的感觉越简单，但要想简单地描述诸多复杂的对象，工具最好拥有一定的复杂性及多样性。

怎么做 多准备一些方法

在设计工具时，要多准备几个方法。要以“简单的事情简单做，复杂的事情也能做”为目标。

对使用者来说，选项增多意味着需要花更多的时间去熟悉工具。不

过熟悉之后，使用者就能灵活使用工具了。这时，他们就知道如何从多个选项中选出最符合当前情况的选项了。

工具由于其性质，使用的时间远比编写的时间长。因此，用在完善工具上的投资是有杠杆效应的。即便在设计上花费了很多时间，只要能做出一个好的工具，我们就能得到无穷大的回报。

扩展 TMTOWTDI 与简单性

对软件而言，简单性的优先级很高。不过，简单性应该放在哪里呢？对于这个问题，我们就需要从整个软件的角度来考虑了。

TMTOWTDI 认为，在工具类软件的情况下，工具方应承担复杂性，简单性应留给用户方。

在其他情况下我们也不应该一味追求简单性。在某些情况下要对复杂性持有宽容的态度，以优化生态系统整体为先。

出处

[1] 拉里・沃尔，等．perl 语言编程 [M]. 何伟平，译．北京：中国电力出版社，2002.




第 6 章　手法

~ 程序员的工具箱 ~

只有脚踏实地进行实践才能从义务的
重担下解放我们自己。

——歌德






曳光弹

英 语 Tracer ammunition

是什么 留在最终代码里的“骨骼代码”

曳光弹是一种会发光的子弹，能让人看清其飞行轨迹。在弹带上，每隔几发普通子弹就会装填一发曳光弹。

曳光弹会立刻给我们反馈。曳光弹发射后，会像烟花一样描绘出从枪口到着弹点的弹道，帮助我们在射击过程中修正瞄准位置。

因为曳光弹与普通子弹在同一环境下发射，所以外部因素造成的二者的偏差被抑制到了最小。曳光弹能命中目标，意味着普通子弹也能命中目标。

同样，在编程过程中，我们也需要打出“曳光弹”来捕捉目标。

编程中的曳光弹指先对那些想要优先检验的部分进行编程。此时我们可以编写简单的端对端软件。软件只要能在实际环境下运行并进行检验即可。

这里编写的其实是成品软件的框架。编写的代码都是正式代码，它们之后会通过项目不断变得丰满，最终存留在成品软件之中。

为什么 在黑暗中照亮道路

曳光弹由于能与普通子弹在同一环境、同一制约条件下发射，所以所获取的信息的准确性较高。由此，我们便可以立刻获得正确的反馈，准确捕捉到目标。从成本的角度来看，这也是比较廉价的一种方案。

要想在软件开发中获得同样的效果，就需要以迅速且能看得到的形

式多次提示与软件最终形态有关的决策。所以，我们需要有正式的软件。

特别是在从零开发软件时，程序员必须像枪手一样在黑暗中锁定目标，因为在没有系统的情况下，用户的需求会变得模糊。程序员自己在开发软件时也要面对生疏的算法、开发技巧、语言和库等。而且，如果项目耗时过长，情况与目的等也很有可能发生变化。

在这种情况下，使用曳光弹能获得以下好处。

- **能得到用户的反馈**

  在使用曳光弹的情况下，我们可以提早把软件展示给用户。只要交流得当，我们就能从用户那里得到有用的信息。

  用户能告诉我们功能方面有哪些不足。看着软件一步步完成，他们也会感到高兴。用户参与到项目之中，对项目的感情也日益加深。在此过程中，用户还会告诉我们软件距离最终目标还剩多远。

- **能尽早为程序员提供发挥本领的舞台**

  没有什么比空无内容的说明书更打击程序员工作的积极性了。在这种情况下，我们仿佛置身于缺氧的环境中。

  尽早构思并实现软件的端对端交互，能提早为程序员提供发挥本领的舞台，提高全员的工作效率。

- **能更快、更正确地进行测试和调试**

  在使用曳光弹的情况下，我们能获得用于整合代码的平台。

  软件完成端对端连接，就意味着搭建好了能整合代码的环境，当日完成单体测试的代码可以立刻整合。也就是说，整合工作可以分散到每天，甚至可以一天进行多次。

  这样一来，我们就能立刻发现新的修改所带来的影响，修改的相互作用也限制在了一定范围内。因此，调试与测试能变得更快和更正确。

- **确保软件可以演示**

  没人知道项目的利益关系者哪天想看演示。在加入了曳光弹的开

发中，我们能获得可以随时演示软件的环境。

● **进展更加明确**

在加入了曳光弹的代码开发中，我们能以用例为单位进行开发。这样一来，生产率报告和提供给用户的进度报告都会变得更加明确。另外，由于各个开发单位较小，所以不会出现巨大且不可拆解的代码块。

怎么做　先编写能够运行起来的基础部分

先编写基础部分，然后慢慢添加其他内容。

基础部分的代码要保持在最小的状态。不过要注意，这些代码必须能运行起来才行。它们最后会成为软件最终形态的“骨骼”的一部分。也就是说，这些代码不是用完就扔的。

另外，在向基础代码添加内容时，一定要检查添加的内容是否偏离了目标。要时常对添加的内容进行检查，并根据需要修正路线。在这种开发模式下，只要捕捉到目标，后面的添加操作就会变得非常简单。

扩展　与原型的区别

曳光弹的作用与原型的作用类似，但二者有着明确的区别。

● **原型**

原型的作用是验证软件的概念或最终形态是否在理解上存在偏差。因此，确认过概念之后，我们会把创建的东西全部抛弃，然后根据获得的信息以正确的形式重新搭建软件。

假设我们正在开发一个供配送中心使用的软件。软件中有一个用于将大小不一的货物巧妙装车的功能。此时有两个问题需要解决，一个是难以设计出一目了然的用户接口，另一个是用于得出最佳装载方式的算法太难。

在这种情况下，首先我们要使用能简单搭建画面的开发工具创建出用户接口的原型。

原型中需要有画面显示以及一些模拟操作，只要能让终端用户确认使用时的感觉即可。当用户按下按钮时，相应功能不必真正运行起来。我们要通过原型找到能让用户满意的用户接口，之后将原型抛弃，重新编写代码。

同样，我们也要创建多个装载算法的原型。在需求上与终端用户达成一致意见后，按照定下来的算法重新编写代码。

● **曳光弹**

曳光弹能体现出软件内部是如何相互协作的。它能为程序员提供可以在今后一直使用的构架，能为用户展示使用效果。

如果将曳光弹用于上述示例（开发供配送中心使用的软件）中，就是创建并整合简单且能运行的用户接口和装载算法。虽然此时的代码还很粗糙，但只要把软件中的所有模块连接起来，用户和程序员就能获得基础框架。

随着开发的进行，新的功能会不断添加到框架中。简陋的临时代码会被逐渐替换成正式代码。不过，框架本身会存留下来。当初创建出来的代码是怎样运行的，软件就怎样运行。

总而言之，原型生成的代码是用完就扔的。这些代码只是整体的一部分，是被临时拿来使用的。而曳光弹是最小限度的可运行的正式代码，是系统最终形态的“骨骼”的一部分。这个差异十分重要。

原型的真正用途在于“学习”。原型的价值不存在于生成的代码中，而存在于我们得到的教训里。

原型可以说是发射曳光弹之前的“侦查活动”或“谍报活动”。

关联信息 原型、样品与试制品

原型的真正用途在于“学习”。这种“学习”包含两个方面的内容。

- **用作样品的原型**

  这类原型主要用于帮助我们准确掌握用户的需求。我们可以通过让用户使用样品来确定需求。

- **用作试制品的原型**

  这类原型用于帮助我们摸索开发方法。

  我们在开发一种全新类型的软件时会遇到很多未知的东西，风险很大。因此，为了弄清具体的开发方法，我们需要用到原型。

出处

[1] 安德鲁・亨特，大卫・托马斯. 程序员修炼之道 [M]. 周爱民，译. 北京：电子工业出版社，2011.

相关图书

[1] 国友義久. 効果的プログラム開発技法 [M]. 東京：近代科学社，2009.

[2] 杰瑞德・理查森，等. 软件项目成功之道 [M]. 苏金国，王少轩，译. 北京：人民邮电出版社，2011.



契约式设计

英 语 Design by Contract

简 写 DbC

是什么 调用方与被调用方的契约

函数是用来做某项工作的。在开始某项工作之前，函数要先“预想”工作完成后的状态，在工作结束之后，要对当前状态履行某种“约定”。

这里的“预想”称为“前置条件”，是函数调用方应当遵守的契约。而“约定”是“后置条件”，是函数方应当遵守的契约。

这种函数与函数调用方互相缔结契约的编程方式，称为“契约式设计”。

在契约式设计中，函数调用方需要根据契约，在调用函数之前满足前置条件。传递正确的参数是函数调用方的责任。

另一方面，函数方要满足后置条件。在函数被调用之后满足相应条件是函数方的责任。

如果某项契约条款未履行，则启动补救方案。具体来说就是抛出异常、终止软件运行等。

为什么 尽早发现理解上的偏差

严格遵守契约能获得以下好处。

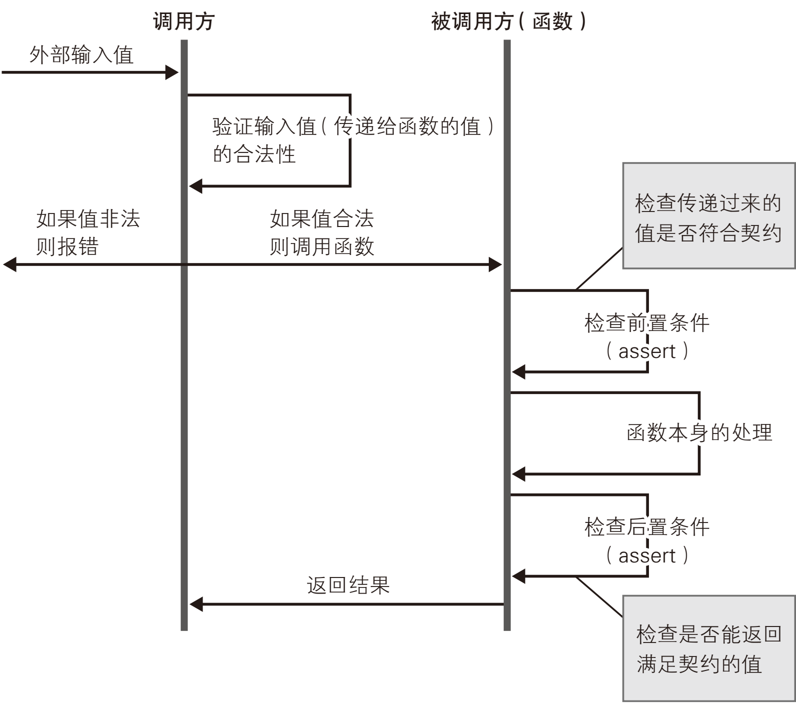
- 能保证代码的正确性。正确的代码只会按照要求做事
- 能使代码保持简洁。函数的处理代码可以在假设其满足前置条件的前提下编写

- 容易提早发现问题。出现无法履行契约的情况就意味着会有故障发生，我们不能视而不见。出现问题时立刻让系统崩溃能降低排查故障的难度

怎么做　通过注释和断言缔结契约

我们要把契约内容事先写在函数的注释里。

检查契约是否被履行的代码要以断言的方式编写。用断言检查前置条件与后置条件，一旦发现契约未被履行则立刻终止软件。契约式设计的处理流程如下图所示。



我们要注意以下几个地方。

- **函数方不调整参数**

  被调用的函数方不可以调整传递过去的参数。比如函数方在检查参数时发现参数不符合前置条件，此时就不能将其转换为符合前置条件的参数。调整参数的处理应该在调用函数之前（上图“验证输入值的合法性”之后）由调用方执行。被调用的函数方应当根据契约放心地使用参数。这样能保证代码简洁。

  不过，为应对参数不合法的情况，函数方还是应该通过断言检查契约的履行情况，提高调试的效率。

- **函数方的断言不用在用户的输入检查中**

  最具代表性的外部输入值当属用户在屏幕上输入的值。不过，前置条件只是函数与函数调用方的契约，不是函数与用户输入的契约。没有履行契约，就表示不该发生的事情发生了，这与用户的输入错误有本质上的区别。这两个概念一定不能混淆。用户的输入检查需要在调用函数之前（上图的“验证输入值的合法性”）由函数调用方完成。

- **预想应严格，约定应宽容**

  如果要保证函数能接受所有的值，并且在任何时候都能返回正确结果，就需要编写大量的代码。为防止此类事情发生，我们要对处理前的预想（前置条件）有较高的要求，函数返回结果时应尽量减少约定（后置条件）。

  尽量少写代码。我们追求的是“懒惰的代码”。

扩展 不变式

在面向对象的契约式设计中，除了函数的前置条件和后置条件，还有一项契约是“不变式”。

这项契约保证类对于使用类的一方来说一直为真。虽然这一点无法在函数内部处理的过程中得到保证，但在函数执行结束，控制返还给调用方的时候，不变式能得到履行。

类方要保证该契约能被履行。

关联信息一 断言

在代码中，预想之外的东西用断言来表现。

大部分编程语言有 assert 这样一个检查表达式真伪的功能。我们要把应满足条件的表达式传给 assert 的参数。

断言为真，则表示所有代码都在顺利运行。断言为假，则表示从代码中检测出了预想之外的错误。当断言为假时，软件会停止运行并返回信息提示用户发生了不该发生的事情。

断言用于让开发过程中的代码暴露出相互矛盾的条件、预想之外的状态、传递给函数的非法值等。断言能够让违反契约的行为以及修改代码时混入的错误迅速暴露出来，在规模巨大且复杂度较高的代码和对可靠性要求较高的代码中特别有效。

使用断言时有几点需要我们注意。

传递给断言的表达式不能有改变变量的值之类的副作用。断言是否被执行会根据编译方法而发生改变。一旦变量的值发生变化，运行差异就会给我们造成困扰。

断言不应该包含在发布版的软件中。断言是用来进行开发和维护的。断言的信息并不是给终端用户看的。软件中包含断言会影响软件的性能。所以在发布版的软件中，一定要记得将断言从编译对象中剔除出去。

关联信息二 继续执行代码不如让软件停止运行

在代码的执行过程中，一旦发生了不该发生的事，应尽早让软件停止运行。

当发生不该发生的事时，代码就陷入了无法继续运行的状态。强制运行只会使情况变得越来越糟。损坏的数据一旦进入重要的数据库中，将会酿成无法挽回的后果。

考虑到这些风险，在出现可疑情况时，软件应立刻停止运行。相较

于带着故障继续勉强运行，立刻停止运行的做法危害更小。

所以在发生问题时，应尽快完成释放资源、输出日志等最低限度的处理，然后尽早停止软件的运行。

出处

[1] Bertrand Meyer. Object-Oriented Software Construction[M]. Upper Saddle River: Prentice Hall, 1997.

相关图书

[1] 安德鲁・亨特，大卫・托马斯. 程序员修炼之道 [M]. 周爱民，译. 北京：电子工业出版社，2011.



防御性编程

英 语 Defensive programming

是什么 防患于未然的程序设计

我们在编程的时候不要想当然。

防御性编程与开车时的防御性驾驶是同一种思路。

在采取防御性驾驶这一驾驶方式的情况下，我们总抱有一种不知道其他驾驶员会做出什么事情的心态。也就是说，自己不认为驾驶的过程是百分之百安全的，觉得中途可能会发生什么事。这样一来，当其他驾驶员做出一些危险的行为时，自己就能做好充分的准备不受伤害。即便过失在其他驾驶员身上，自己的命也要由自己来保护。防御性驾驶体现的就是这样一种心理。

与此类似，当函数接收到非法数据时，即便问题出在其他函数身上，我们也应准备好“防御性”的代码以避免函数受到损害。为此，编程时要注意以下几点内容。

- **确认外部代码传来的数据输入值（检测“预想之内的错误”）**

  在从文件、用户接口、网络以及其他外部接口获取数据时，要确认数据是否在合法范围内。比如检查数值是否在有效范围内、字符串的长度是否符合规定等。

  尽量在较早的阶段检测出无效输入。检测出无效输入后要迅速对其进行适当的错误处理。

- **确认参数的值（检测“预想之外的错误”）**

  确认其他函数传来的参数的值。与检测外部代码传来的数据不同，这里如果检测出无效输入，就意味着代码存在 bug。

  我们可以使用断言确认参数，在发现非法值时立刻停止程序。

为什么　开发与运维中的“安全驾驶”

就像防御性驾驶能使驾驶更安全一样，“防御性编程”可以让编程更安全。

安全表现在如下两个方面。

- **开发中的“安全驾驶”**

  提早发现非法数据能提升调试的效率，因为提早检测出非法数据，并以明确的形式进行通知，可以帮助我们立刻找到出现问题的地方。这样一来，代码的调查与修改都变得非常容易。

  反过来，如果没能提早检测出非法数据，那么故障就会蔓延到其他地方，这时我们就需要花费更多的时间来寻找根本原因。

- **运维中的“安全驾驶”**

  尽早处理非法数据能防止运维中出现的问题进一步扩大。在较早的阶段处理掉问题，能防止问题的蔓延。

  错误如果处理得不彻底就会蔓延到其他处理中，问题会变得越来越大。特别是当错误的数据进入软件深处时，软件的运行可能会发生错误，或者错误的数据会进入数据库中，这将造成无法挽回的后果。

  其中最棘手的当属安全问题。黑客在入侵系统时，喜欢利用没有彻底处理错误的地方。可见，不完备的错误处理也可能会成为安全漏洞。

怎么做 路障战术

我们需要采用“路障战术”。建立路障，将损害控制在一定的区域内。

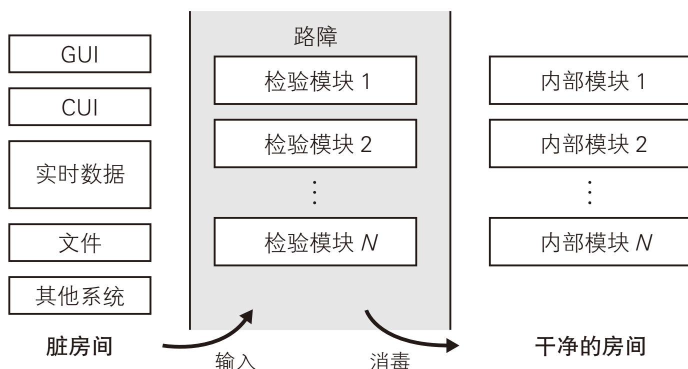
船体由多个相互隔离的区域组成，这与路障战术是同一种战略思想。即便船撞上冰山，船体破损，只要隔离破损的区域，整个船体就不会有沉没的危险。

另外，建筑物中的防火墙与路障战术也有异曲同工之妙。防火墙的作用在于防止火势蔓延。

为了在代码中建立路障，我们需要将特定的接口用作安全地带与非安全地带的分界线。检验通过这条分界线的数据，一旦发现非法数据，立即采取适当的措施。

这就好比手术室，所有东西都必须经过消毒才能拿进去。因此，通过大门进入手术室的东西都是安全的。

以门（＝路障）为界，分界线的左侧是“脏房间”，右侧是“干净的房间”。



在代码设计中，我们要明确“哪些东西可以进入手术室”“哪些东西不能进入手术室”，以及“门的位置”，也就是对安全地带里面的模块、安全地带外面的模块和在中间负责消毒的模块进行分工。

扩展一 断言与错误处理的区别

用断言应对预想之外的错误，用合适的错误处理方法应对预想之内的错误。采用路障战术之后，我们就能明确区分这两种应对错误的方法了。

在路障外面对数据进行预测是一种危险的行为。因此，路障外面的模块适合使用错误处理的方法。

而传到路障内的数据在通过路障之前就已经“消过毒”了。如果路障内的某个模块检测到了非法数据，那一定不是数据出现了错误，而是代码出现了错误。因此，路障内的模块不能采用应对数据错误的错误处理方法，而要采用断言让代码自身来检查代码执行过程中是否有问题。

扩展二 错误处理的变种

对于预想之内的错误，不同的情况有不同的处理方式。具体来说有以下几种处理方式。

- **返回无害的值**

  在确认某值无害的情况下，返回该值。

  比如在数值计算的情况下返回 0，在字符串计算的情况下返回空字符串，在指针计算的情况下返回 NULL。

- **使用下一个数据**

  在处理一连串数据的情况下，返回下一个有效数据。

  以从数据库读取记录为例，如果记录无效，则一直读取，直到发现有效记录。

- **返回和前面一样的值**

  如果不会对结果造成重大影响，则返回和前面一样的值。

  以 1 秒内读取 100 次温度计的代码为例，如果其中有一次读取失败，在这种情况下，即使返回失败前最后一次读取的值，也不会有什么问题。

● **使用近似值**

在满足一定的严密性的前提下，返回近似值。

比如在能显示 0℃ ~100℃的温度显示画面中，温度低于 0℃时显示 0℃，高于 100℃时显示 100℃。

● **在日志中记录警告信息**

在日志文件中记录警告信息后继续执行处理。

当发生微小的错误时，忽略错误继续执行处理有时是一个很好的选择。不过，发生过的错误一定要记录下来。

● **返回错误**

为了调用上游函数来处理错误，我们要将检测出来的错误记录在报告中。

在这种情况下，决定让代码的哪个部分负责处理错误，哪个部分负责报告错误就变得至关重要。

我们可以使用模块的状态变量、函数的返回值，或者通过抛出异常来报告错误。

● **调用错误处理函数**

错误处理要交给共同的错误处理函数来完成。

将错误处理的责任一元化能降低调试的难度。不过，这个一元化的功能会使代码整体产生较高的耦合度。因此，如果想把一部分代码用到其他系统中，就需要连同错误处理算法一起“搬家”。

● **显示错误信息**

在发生错误的地方显示错误信息。

将错误处理的开销抑制到最小。不过，由于信息会分散在软件各处，所以创建具有统一性的用户接口、区分用户接口与其他部分、将软件转换为其他语言等工作变得难以实施。

- **终止处理**

  检测到错误后终止处理。

  这个方法对重视安全性的软件来说非常有效。在关键任务系统中，比起带着错误继续处理，很多时候重新启动程序会比较好。

- **各部分选择最合适的方式处理错误**

  选择何种方式处理错误，由负责设计与实现错误发生部分的程序员来决定。

  这给了程序员很大自由，但从软件整体来看，错误处理将失去统一性。

扩展三　错误处理中的“正当性”和“坚固性”

错误处理中有“正当性”和“坚固性”两种思路。

正当性指一定不返回不正确的结果。与其返回不正确的结果，不如什么都不返回。

而坚固性指为了让软件继续运行而不择手段。即使会产生不正确的结果，也要让软件继续运行下去。

以哪种思路为先，就要看软件的目的是什么了。

重视安全性的软件要以正当性为先。与其返回错误结果，不如直接停止软件。以医疗相关的管理软件为例，相较于返回错误结果继续处理，通知错误并停止软件更能防止重大事故的发生。

而对于提供给用户的软件，坚固性就要优先于正当性了。以文字处理软件为例，比起软件突然关闭导致大量宝贵的输入数据丢失，带着错误继续运行所造成的损失更小。

扩展四　将输入数据转换为正确的格式

通常，当外部传来数据时，软件需要对其进行“消毒”。此时，我们要让数据在输入的瞬间就转换成正确的格式。

一般情况下，输入数据会以字符串或数值的形式传递。这些值有时被映射为 Yes、No 等布尔型，有时被映射为 Red、Green、Blue 等列举型。

格式不定的数据逗留在代码中会使情况变得复杂。这时，有人可能就把 Yes 当作颜色输入进去，使软件发生冲突。记住，输入的数据一定要立刻转换成正确的格式才行。这是基本中的基本。

扩展五 不忽视错误代码

不忽视错误代码是防御性编程的铁则。

即使函数返回错误代码，接收方也有可能会忽视掉它。但是，我们一定要养成评价函数返回值的习惯。即便某个函数在理论上不会发生错误，保险起见我们也要对其进行检查。因为防御性编程的目的就是防止预料之外的情况出现。

自己编写的函数不能忽视错误，系统函数同样不能。每次进行系统调用都要检查错误代码。

发现错误之后，要在日志中输出错误编号以及错误的详细内容。

关联信息 “到语言中”编程

“在语言中”编程的程序员会把自己的想法限定在该语言直接支持的结构中。因此，如果使用的语言比较初级，程序员的想法就会受到语言的限制，变得初级。

而“到语言中”编程的程序员会先思考自己想表达什么样的思维逻辑，然后将思维逻辑套用到本次使用的语言中，决定具体的表达方法。

所以，我们不应“在语言中”编程，而应“到语言中”编程。

不可否认，很多时候语言中并不包含我们想使用的表达方法。但是，一个好的想法不应该被语言的机制束缚，而应该突破语言障碍得到实现。

我们可以尝试用非标准库进行补充，或者借助预编译来实现标准

化。如果再不行，还可以考虑使用编码规则来对编程行为进行约束。

假设我们想在代码中表达“契约式设计”或“防御性编程”的思路，但语言并不具备断言功能。在遇到这种情况时，我们不能把眼光局限在语言之中，放弃自己的想法，而应该寻找合适的开源库，或者自己编写宏，想方设法将思路实现到语言中去。

出处

[1] 史蒂夫·迈克康奈尔. 代码大全（第 2 版）[M]. 金戈，汤凌，陈硕，张菲，译. 北京：电子工业出版社，2006.

相关图书

[1] 凯佛林·亨尼. 程序员应该知道的 97 件事 [M]. 李军，译. 北京：电子工业出版社，2010.

[2] 理查德·蒙森·哈裴尔. 软件架构师应该知道的 97 件事 [M]. 徐定翔，章显洲，译. 北京：电子工业出版社，2010.

[3] 皮特·古德利弗. 编程匠艺——编写卓越的代码 [M]. 韩江，陈玉，译. 北京：电子工业出版社，2008.



内部测试

英　语　Dogfooding
Eating your own dog food

是什么　试尝软件的“味道”

我们要试着用一下自己开发的软件。

还未开发完成或者刚刚发布的软件，其质量还不够好，软件本身并不完美。这时的软件可能“味道”不是很好。

即便如此，它也是我们自己开发的软件，是拿给别人用的东西。所以我们自己要先“品尝”一下软件的“味道”。

为什么　从用户的角度出发

对于自己开发的东西，开发团队总会下意识地回避故障。

而以用户的身份使用软件，能帮助我们站在用户的角度思考问题。

这样一来，我们就能发现测试时没有发现的缺陷。在用户使用软件之前发现软件的故障并对其进行修正，能提高软件的稳定性。

另外，以用户的身份使用软件，也能发现一些没有用的功能以及其他可能会用到的功能，还能发现哪些功能用起来不是很顺手。这样一来，我们就能发现一些站在程序员的角度无法发现的需要改善的地方，解决功能过剩与功能不足的问题，改善用户体验，提升软件的魅力。

怎么做　以用户的身份使用软件

在发布软件之前，我们要作为用户使用一下软件。不是去模仿用

户，而是真正地作为用户去使用软件。

软件在发布之前通常质量较差，可能会影响日常工作。但是和被用户抛弃的风险相比，这算不了什么。在发布之前以用户的身份使用软件，能确保软件的稳定性，避免引起用户的不满，还能改善功能，吸引更多的用户。我们没有不做之理。

当然，发布之后也要坚持使用自己的软件。既然打着“方便好用”的旗号把软件交给用户，就应该亲自使用软件证明给他们看。

出处

[1] 乔尔・斯波尔斯基. 软件随想录 [M]. 杨帆，阮一峰，译. 北京：人民邮电出版社，2015.



橡皮鸭调试法

英 语 Rubber ducking

别 名 小黄鸭调试法

黄鸭除虫法

是什么 借助“说明”完成调试

橡皮鸭调试法是一种调试方法。

在编程的过程中，我们要把出现的问题或者有问题的代码说给“某个人”听。这样一来，我们就能找到问题出现的原因，并对问题予以解决。

这里的“某个人”可以是物品，比如放在浴缸里的橡皮鸭等，只要能一边点头一边听我们说话即可，其自身不需要讲话。

这一方法能产生立竿见影的效果。有时候，我们刚开始说明就能注意到问题所在。

这个方法简单且廉价，但效果出奇地好。

为什么 促使自己解决问题

橡皮鸭调试法因为操作起来太过简单，所以很多人觉得没有多大意义。但事实并非如此。

在向他人说明问题之前，我们首先要通读代码，明确其中隐藏的假定条件。我们在把这些假定条件说出来的时候，会对问题产生新的见解，这些新的见解会成为解决问题的突破口。

另外，在说明的过程中，一旦说到有问题的代码，我们就会发现该部分“无法说明”，或者“说明的内容与代码不符”，从而发现存在缺陷的地方。

只要像上面那样按部就班地对问题进行说明，问题出现的原因就能浮现出来。

怎么做　向没有生命的物品说明问题

在解决问题时，如果遇到瓶颈，可以找一个倾诉对象说明一下问题。

这个对象可以是橡皮鸭等没有生命的物品。重要的是“说明”这一行为。

比如某大学信息技术系的服务台旁边总会备一个泰迪熊玩偶。该学校规定，学生在遇到奇怪的故障时，必须先向这个玩偶说明一遍问题再咨询工作人员。该做法取得了很好的效果。

但话说回来，最好的说明对象仍然是程序员同事。因为我们在向他们说明时最具真实感和临场感。我们自己无法解决问题的时候，对方还会给出建议。

不过，同事的时间也是宝贵的。从团队优化的角度来看，在使用橡皮鸭调试法时，以物品为倾诉对象的效果更佳。如果无论如何都要与同事交流，应先说明主旨，取得对方的同意。

出处

[1] 安德鲁・亨特，大卫・托马斯．程序员修炼之道 [M]．周爱民，译．北京：电子工业出版社，2011.

相关图书

[1] 史蒂夫・迈克康奈尔．代码大全（第 2 版）[M]．金戈，汤凌，陈硕，张菲，译．北京：电子工业出版社，2006.



6.6 语境

英 语 Context

是什么 在语境中对话，结合语境思考

语境又称为上下文，指周围的情况与背景。

编程时，我们要从两个侧面充分利用语境这个概念。

- **用于代码的读写**

  代码的读写是写代码的人与读代码的人之间的交流。写代码时要注意给读代码的人创造语境。为此，我们要自上而下地编写代码。

  具体来说，就是让模块名等大块代码的名称足以说明这块代码是做什么的。模块中的函数名也要以模块名为语境，并体现出函数做了什么样的处理。

  这样一来，对读代码的人而言，模块的名称和函数的名称就相当于图书的章标题和节标题，用于提醒人们当前阅读的是哪里的代码。这对读代码的人非常有帮助。

- **用作思考的工具**

  编程是一种解决问题的工作。

  只关注单独的要素不足以解决问题。事物不是单独存在的，它们与周围的情况息息相关，有自身存在的原因。所有要素相互关联，相互牵扯。把这些东西视为语境纳入思考的范围，能帮助我们解决问题。

为什么 防止对话与思考失去方向

给说明附上语境，能将零散的信息联系在一起。于是，代码的意义变得更加连贯，代码的可读性变得更高，读代码的人也不容易让自己的思考失去方向。

代码必须能让读代码的人轻松理解代码正在做什么。没有语境，对方就失去了理解的基础，信息传递的效率必然不高。

编程高手都会使用语境。他们会从直觉上联系语境，并结合自身的经验，从现有的解决方案中选择一个合适的方案来实施。当然，这里的直觉是有根据的直觉。毕竟这些编程高手能读懂代码的“氛围”，所以他们不用按照条条框框来进行操作。

而初学者常常忽视语境，按照条条框框工作。他们只会按照指示做事，因此最终展示出来的成果会与编程高手存在差距。速度和质量都相去甚远。而且初学者一般无法应对未知情况，所以在某些情况下即使花再多的时间也不会得到很好的结果，此时初学者与编程高手的差距就无法计量了。

编程高手之所以能熟练运用各种模式，并不是因为他们知道的模式多，而是因为他们知道模式应该在何种情况下使用、以何种目的使用。编程高手看的就是语境。

怎么做 展示语境

在写代码时，要先将语境展示出来。这样一来，读代码的人就能对具体情况做出判断，第一时间掌握这些代码的用途。

在读代码时，应先获取语境，然后在此基础之上理解代码。

在设计代码时，应效仿高手的思考方式。要成为高手，最有效的方法就是模仿高手的行为。具体来说，就是在解决问题时，以语境为前提进行思考。我们没有高手丰富的经验，但是可以模仿他们的思考方式。我们要训练自己，让自己认识到语境的存在，挖掘问题的语境，根据情况和目的进行判断。

另外，要让所有工作都远离那些独立于语境的绝对规则。为了摆脱初学者的身份，成为高手，我们要将重心放在直觉上。久而久之，我们就会像系统思维所提倡的那样，自己成为系统的一部分，主动地去解决问题。

此外，我们还要以“着眼大局，着手小局”为工作战略。要先结合语境看整体，完成构想，然后一点一点进行实践。这样，我们就能做成一件大事。

扩展一 语境与工作委托

在委托别人写代码的时候，语境的相关观点同样有效。

我们在让编程高手写代码时，不需要设置什么条条框框的规则，只要将足够搭建语境的信息告诉对方即可，之后便可任其自由发挥。这样编程高手就能做出最好的作品。

相反，用规则束缚编程高手，并不是什么明智之举。让一大群马聚在一起奔跑，马一定跑不开。况且人还有情绪方面的问题。多余的规则会降低人的积极性，闹不好还会把人逼出团队，使团队的战斗力大打折扣。

我们要告诉高手最终要达到的目标，这一点并不限于编程。

比如对于清扫能手，你想让他去打扫浴室，这时只要告诉他把浴室打扫干净即可。清扫能手看到浴室的具体情况，就会选择合适的工具，然后结合性价比、时间等因素，找出最佳的解决方案并付诸行动。如果浴室已经脏到无法打扫干净了，清扫能手可能会跟你谈重新装修的事情。总而言之，他们会采用各种手段，倾注全力来达到目的。

换成初学者的话，这种方式就行不通了。如果只提供“把浴室打扫干净”这样一个抽象度较高的信息，他们就会无从下手。因此，你需要下达一些具体的命令，比如先用刷子刷一遍浴室，等等。

然而初学者并不理解真正的目的，他们只会去刷一遍浴室。而且就算刷的时候因力道不足或次数较少而没刷干净，也会向你报告：“我（按你说的）刷完了。”你过去一看，浴室确实刷了一遍，但它并没有变干净。

接下来你将陷入给出具体方案的泥潭。“用这么大的力气刷”“刷三遍以上”“用清洁剂”“清洁剂用这么多”“再用上抹布”……每做完一步你都要去确认一下，然后发出下一个指示。

在最坏的情况下，你还可能会落一个“指示有问题”“尽挑毛病”的埋怨。再加上初学者技术较差，工作失误较多，能做的事情有限，使得指挥也变成了一项苦差事。

这样想来，高手非常难得。我们要努力让高手把专业技能留在现场。要让他们在工作的时候有一个好心情，不会产生离职的想法。

另外，要想培养高手，让高手维持高手的水平，必须把高手放在一线。我们常看到高手被摆在管理职位上，这其实是一种浪费。

扩展二　语境的传导能力

语境对交流的贡献非常大。

这里做一个简单的实验。下面有两组文字。上面一组不分类别，下面一组是食物。每一个词都由四个字组成，但是文字顺序被打乱了。请大家看着这些文字联想原来的词。

- **无分类**

秋节佳中　→［　　　　］
票高车铁　→［　　　　］
空式调立　→［　　　　］
里万城长　→［　　　　］
月风雪花　→［　　　　］
半更夜三　→［　　　　］
对向象面　→［　　　　］
机气喷飞　→［　　　　］
督门提九　→［　　　　］
命耳面提　→［　　　　］

- **食物**

腰爆炒花 → [ ]

子鸡辣丁 → [ ]

片夫肺妻 → [ ]

醋里糖脊 → [ ]

蒸清鱼鲤 → [ ]

焖虾油大 → [ ]

牛酸肥汤 → [ ]

庆火重锅 → [ ]

丝香肉鱼 → [ ]

豆塌锅腐 → [ ]

试过以后不难发现，第二组词语联想起来比第一组简单。大脑仅仅接收了一个“现在即将接收这类信息”的提示，处理能力就急剧提高。这就是语境的传导能力。

比如，在读书的时候，如果采用自下而上的阅读方式，也就是从单词开始理解，然后是句子，最后是文章，就会给大脑带来极大的负担。

不过，书是有标题的。只要认真阅读标题，我们就能顺畅地理解后面的段落和文章，因为段落所写的内容就是标题所表达的内容。这种确认标题与正文之间关系的阅读方法更加合理。这种阅读方法称为“自上而下型阅读”。

不管是章标题还是节标题，这类引导人们理解后续内容的元素称为“阅读前导”。在前面的实验中，“食物”就是阅读前导。

阅读代码和阅读图书是一样的。代码中的标题就是装载着某一类内容的模块的名称。对于代码中的标题，我们要更用心地去读。反过来，在写代码的时候，为了能让读代码的人构建语境，一定要认真安排阅读前导。

即使从一般语言的角度来说，语境也非常重要，因为单词本身并不具有意义，它们的意义只体现在语境之中。

比如在自家厕所里，孩子焦急地大喊：“妈！”这其实是“帮我拿纸”

的意思。又比如一对恋人在卿卿我我时，其中一方羞红了脸说：“笨蛋!”这句“笨蛋”也不是真的说对方脑袋笨。

语言的内容只有与语境相结合才能获得意义，才有价值，才能传达出正确的意思。

关联信息一　团队要有高语境意识

“高语境”指团队在背景、价值观等很多方面有共同认识的状态。

在高语境文化中，人们沟通起来很有默契，许多事情能做到心照不宣。一般来说，日本属于高语境文化，欧美属于低语境文化。

项目团队进入高语境的状态后，沟通会变得顺畅，沟通成本将大大减少，沟通质量能得到提升，这对产品将产生好的影响。

不过，这个语境并不对新加入的成员共享，所以对于新加入的成员，我们应效仿低语境文化，系统地对事情加以说明。

关联信息二　代码通用化应注意语境

将代码共同的部分函数化是一件好事，但有两点需要我们留意。

- **通用化的部分有时需要分别进行修改**

  就算是仅由几行代码组成的执行相同处理的共同部分，如果各自的内部代码不同，这种共同就是偶然的。二者并不存在依赖关系。即使将这类代码通用化，以后需求发生改变的话，还是要还原通用化的部分，然后各自按照需要进行修改。

  因此，即使代码内有多个地方需要进行同样的处理，只要这些处理各自发挥的作用不同，通用化就不会带来多少优势。

- **修改通用化部分时的影响程度上升**

  代码通用化之后，通用化的代码与使用该代码的各个部分将产生依赖关系。也就是说，通用化的代码即便只修改一行也会影响到

所有使用该代码的部分。

代码相互独立时，相应部分的维护成本小到可以忽略。而通用化之后，每次修改都要花大量时间进行测试。

代码通用化能减少构成软件的代码的行数，但会增强代码间的依赖程度。

因此，在对代码进行通用化时，应注意其语境。即便产生依赖关系，只要通用化的代码和使用该代码的部分处于同一个狭小的语境中，就不会出现问题。此时通用化的优势大于劣势。不过，如果在较大的语境中出现依赖关系，软件的诸多部分就会受到牵连，此时通用化的劣势就超过了优势。

因此，在软件的整体结构尚不清晰的阶段，最好不要轻易推进代码的通用化。应先仔细观察各部分之间的关系，确认语境，总结出可以通用化的部分。

关联信息三　程序员的语境切换

操作系统需要让多个进程共享同一个 CPU。因此，进程会相互穿插，此时就需要保存或还原 CPU 和内存的状态，也就是保存或还原语境。我们把这项操作称为“语境切换”。切换语境的效率是评价操作系统的一个重要指标，但再好的操作系统也无法避免语境切换带来的负担。

编程中也会出现同样的现象。

程序员的脑中装的是与当前问题相关的工作集。在编程的过程中一旦有别的事情插进来，我们就必须暂时弃置当前的工作集，转而加载需要的工作集。也就是在脑中进行语境切换。

在编程中，其他事情的插入造成的语境切换是一种性质非常恶劣、需要我们避免的现象。之所以这么说，是因为程序员的理想是让自己在编程的过程中处于“流态”。

所谓流态，是指让自己沉浸在一件事中，处于一种冥想的状态。进

入这一状态后，人将充满幸福感，同时失去对时间的感觉。觉得没干多长时间，结果一看表，都过去 3 个小时了，这其实就是一种流态的效果。进入这一状态后，编程的效率将大幅提升。

对于编程这种高强度的脑力劳动，“进入状态”是非常重要的，流态不可或缺。甚至可以说，我们只有进入了流态，才能让工作顺利进行。

很可惜，我们无法像摁下开关那样自如地转换自己的状态。要想进入流态或维持流态，我们必须要实现连续工作。一般情况下，我们需要花 15 分钟以上的时间来集中精神。

而妨碍我们连续进行工作的，正是职场中的各种“插入事件”，比如有人过来跟自己说话、有会要开、旁边有人讨论问题、电话响了、窗口跳出未读邮件通知，等等。

在进入流态之前的这段时间里，人对于插入事件非常敏感。因此，我们几乎不可能在嘈杂的职场环境下进入流态。

而且，就算有机会进入流态，一个插入事件就能轻松把我们打回原形。一旦被打回原形，再想进入流态又需要至少 15 分钟的时间。在这期间，我们就相当于什么工作都没做。

假设接一通电话平均需要 5 分钟，进入流态需要 15 分钟。这样一来，每接一通电话就会浪费 20 分钟宝贵的工作时间。要是一天之内来 10 通电话，光消耗在这上面的时间就有半天之多。如果再多 10 通，那一整天就都浪费了。

在这样的职场中，很多时候我们在规定的工作时间内什么都做不出来，因此会选择加班。这就给人一种晚上写代码更有效率的感觉，有人甚至觉得在家写代码才有效率。

更严重的是工作被打断时我们会产生焦躁情绪。每次想集中精力工作时就被人打断，任谁都不会有好心情。明明马上就能沉浸到工作之中了，结果却被强拉回现实。即使想进入自己的世界，一个不怎么重要的事情插进来也会让我们前功尽弃。

这种情况如果持续下去，所有人都会对工作感到厌恶，失去职场的归属感。不仅会影响每天的工作效率，还可能导致人才流失。

然而从管理人员的角度来看，程序员因无法集中精力而感到焦躁不

安并不是什么大问题。因为在管理人员的日常工作中，其他事情不断插进来是家常便饭的事。或者说，他们的工作就是应对一个个插进来的事情。所以管理人员很难理解程序员追求的理想状态。

一旦出现此类情况，就需要对职场进行意识改革和文化改革了。在任何条件下，都要让程序员能在连续的时间内进行工作。如此一来，就能在确保编程时间的前提下，很好地分配时间和调整项目，同时还能营造出避免语境切换的文化。

关联信息四　系统思维与领域驱动设计

系统是相互作用的元素（部分）的集合，它拥有整体的功能。

系统并不是单纯的集合，而是拥有整体功能的集合，这一点非常重要。万物都是相互联系的。物质世界、社会机制、人内心的想法、计算机的逻辑……世间万物组成了一个巨大且相互联系的现实组织。

没有任何东西是孤立存在的。所有东西都是组织的一部分，都是周围情况、上下文等语境的一部分。

因此，有时候某个细小的事物会带来巨大的影响。反过来，我们认为至关重要的东西，其带来的影响并没有想象中那么大。输入和输出不一定成正比，“不对称的影响”正是非线性系统的一个显著特征。

现实世界显然是非线性的。不管我们单独拿出什么东西来看，它都与宇宙的一切相关联。

比如地上长着一棵树，我们可以将其视为一个独立的物体。但实际上这棵树至少与两个循环系统存在关联，这两个循环系统分别是“树叶与空气”“根与土”。树既不一成不变，也不独立存在。

更有趣的是，就连观察某个系统的我们，也不是单纯的观察者。不管我们是否发觉，身为观察者的我们，其实也是系统的一部分。

系统思维这种思考方法就是让我们在考察事物时运用“系统”的概念，统一地、全面地理解对象整体。

系统存在边界。从系统内部的某一结构中取出部分元素进行分析，并不能让我们理解系统整体的功能和行为。因此，为了了解对象的整体

性，系统思维不仅要求我们关注结构中的各个元素，还要求我们关注其中的关联性。

要想用软件解决问题，除了要具备解法（算法）相关的知识，还要对问题有正确的认识。在理解问题时，需要用到系统思维，不单要看问题本身，还要考察问题涉及的领域——语境。

领域驱动设计能有效地将系统思维活用到软件开发之中。这里的领域是指用户知识所及、影响所及、活动所及的领域，也称为业务领域、事业领域或问题领域。总之，这个领域指的是软件准备解决的问题，即“软件的语境”。

领域驱动设计将领域建模后形成的“领域模型”置于软件开发的中心位置。程序员与领域专家组成团队，将通过语言、图和代码实现的“交流”与领域模型一体化，并在此过程中反复深化领域模型。通过这种方式来创造出价值更高的软件正是该设计思想所体现的内容。

软件一旦离开领域，就失去了解决领域内问题的能力，其存在意义将受到质疑。为避免这一情况出现，我们应尽力摸索领域模型，从构建起的领域模型出发驱动所有东西。这样一来，我们就能做出可以反映领域的具有较高价值的软件了。

不过，我们不可能一下子就设计出完美的领域模型。一个优秀的模型必须经历多次发布才能真正成型。我们要在每次发布的过程中加深领域相关的知识，反复研究建模，深化领域模型。

关联信息五 实践知识与整体优化

亚里士多德将知识分为客观知识、技术知识和实践知识三类。

- **客观知识**

  客观知识是普遍的真理，是具有普遍正确性的知识。这类知识不被时间与空间左右，是语境独立的、客观的知识。

  客观知识用通俗的话来讲，就是科学。

- **技术知识**

  技术知识是实用的知识或技术在应用过程中衍生或创造出的技巧。技术知识与客观知识的明显区别在于，技术知识是依存于语境的知识。

  技术知识用通俗的话来讲，就是工学。

- **实践知识**

  实践知识与客观知识和技术知识不同，我们可以称之为“知性美德”。也就是说，实践知识其实是一种具有实践性的智慧。实践知识有时也称为卓见、逻辑和实践智慧等。

  实践知识是一种考虑了语境与具体情况，能够具体问题具体分析，可以在进程中根据需要调整行动目标的智慧。也就是说，实践知识是从实践中获得的高质量共识，是一种通过对价值和逻辑进行高度判断，在时刻变化的语境与情况中采取对整体最有利的行动的能力。

在软件开发中，我们也需要根据自己的能力与价值观，在具体的情况下向用户提供价值。因此我们需要运用实践知识，掌握语境，力求实现整体的优化。

关联信息六 关系主义与故障应对

用于解释存在的思想称为“存在观”。

关于存在观，西方哲学认为首先要有独立存在的“实体”，然后实体与实体之间产生第二层存在，即“关系”。

而认为“关系”是第一层存在，所有实体都是“关系的结节”的思想称为“关系主义”。

关系主义的出现比西方哲学早，佛教哲学中的“缘起”就属于关系主义。存在观曾一度以西方哲学为中心，但在现代，关系主义也渐渐进入存在观的阵地。

关系主义来源于存在观，是一种关注元素与元素的关系性与影响的思维方式。关系主义不细究各个部分，而是重视整体现象。也就是说，这是一种将重心放在语境上的思维方式。

在编程领域，这一思维方式在应对故障方面有很好的效果。发生故障时，我们会下意识地以错误部分的代码为中心进行排查。然而很多时候故障发生的根本原因并不在这里。

因此，我们不能只在代码中寻找原因。我们应将目光转向代码与库的关系、代码与执行环境的关系等语境因素。就算每个部分单独拿出来看都是正确的，只要关系性的部分有问题，软件整体就是欠妥的。

记住这一点，故障排查的成功率将大幅提升。

出处

[1] 安迪・亨特. 程序员的思维修炼 [M]. 崔康，译. 北京：人民邮电出版社，2011.

相关图书

[1] 西村行功. システム・シンキング入門 [M]. 東京：日本経済新聞社，2004.
[2] 亚里士多德. 尼各马可伦理学 [M]. 邓安庆，译. 北京：人民出版社，2010.
[3] 寺田昌嗣. フォーカス・リーディング「1 冊 10 分」のスピードで、10 倍の効果を出すいいとこどり読書術 [M]. 東京：PHP 研究所，2008.
[4] 汤姆・狄马克，蒂姆・李斯特. 人件 [M]. 肖然，张逸，滕云，译. 北京：机械工业出版社，2014.
[5] 埃里克・埃文斯. 领域驱动设计 [M]. 赵俐，盛海艳，刘霞，译. 北京：人民邮电出版社，2010.
[6] 凯佛林・亨尼. 程序员应该知道的 97 件事 [M]. 李军，译. 北京：电子工业出版社，2010.




第 7 章　法则

~ 编程的反模式 ~

人的聪明才智不在于经验的多少，
而在于应用经验的能力的强弱。

——萧伯纳






布鲁克斯法则

英 语 Brooks' law

是什么 增员等于“火上浇油”

对于开发进度滞后的软件开发项目，如果为了赶进度而在开发后半程添加人手，反而会使延迟情况进一步加重。

在项目尾声，当我们发现产品无法如期交付时，常会投入更多的人手。但这种做法只会火上浇油。

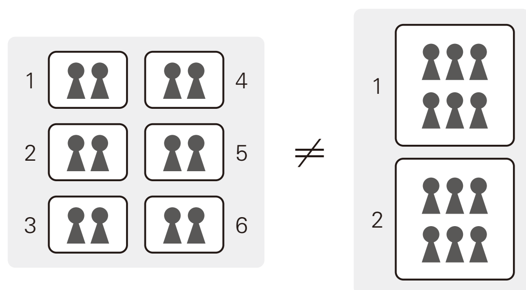
为什么 人数和月数是无法交换的

项目的工时是用人数和月数换算的，也就是几个人用几个月完成某个项目，所以用“人数 × 月数”来计算项目工时。

这里要注意的是，该乘法运算与数值的乘法运算不同，人数和月数不能调换。也就是说，“人数 × 月数 = 月数 × 人数”的式子是不成立的。

比如一个 12 人月的项目，客户要求 6 个月内开发完成，那么我们只要投入 2 人即可。如果人数和月数可以调换，那么当客户说这个项目比较急，需要在 2 个月之内完成时，我们只要投入 6 个人就行了。

然而在现实中，“6 × 2”和“2 × 6”并不相同。2 人工作的效率与 6 人工作的效率不可同日而语。



理由如下。

- **因存在依赖关系而产生额外的负担**

  如果每个人的工作相互独立，那么在人数是原来3倍的情况下，生产效率也会变为原来的3倍。

  然而一般来讲，工作分割之后，各项工作之间会产生依赖关系。如此一来就会产生一些新的负担，如任务的分割、各项确认工作的出现以及通信路径的增加等。

  即便追加人手，这些额外的负担也会拖慢项目的进度。

- **培训新人会占用一定时间**

  在追加人手时，为了能让这些人发挥作用，必须让他们学习当前项目固有的各种知识、信息以及技术。也就是说，要花时间对新人进行培训。此外，负责培训的人是同一个项目内的成员，这就导致新团队的整体生产效率下滑。

  在新人真正发挥作用之前，整个项目的进度都是滞后的。

怎么做　重新制订时间表

无条件地投入更多人手来赶上进度是一种不明智的做法。

强行给当前成员增加负担也只会对项目造成损害。

进度滞后最好的解决方法是重新制订时间表。在此过程中，要与用户做好协调，同时决定各个功能的优先程度，进行阶段式发布。

扩展一 生产与生孩子

在项目开发中，增加程序员的数量并不代表能加快处理工作的速度，缩短开发时间。

在这一点上，开发的生产效率与生孩子很像，因为就算把 10 个孕妇凑到一起，妊娠过程也不可能缩短到一个月。

这个比喻生动形象地展示了生产效率不是单纯的劳动力相加。

扩展二 人与人也不可交换

前面说过，人数与月数不可交换。从某种意义上讲，人与人也是不可交换的。

一个程序员离开了，并不是再补一个程序员就行。之所以这么说，是因为程序员的水平参差不齐。

在物理空间内的生产效率方面，有能力的人与没能力的人之间的差距最多也就几倍。但像程序员这种以信息空间为主战场的人，由于不受物理方面的制约，各个程序员之间的生产效率有很大的差别。据说能差 30 倍。

不过，“同样的时间内能写出多少代码”这种生产效率上的差距并不是造成上述现象最根本的因素。某些方面的差距更根本且更巨大。

比如以下几个方面。

- **有能力 / 没能力**

  有些人写出的代码能用，有些人写出的代码不能用。这是一个有与无的比较，计算差距已经没有意义了。

- **bug 多 /bug 少**

  有些人写出的代码没有 bug，有些人写出的代码到处都是 bug。二者的维护成本会出现巨大的差别。

- **执行速度快 / 执行速度慢**

  有些人写出的代码执行速度快，有些人写出的代码执行速度慢。

  代码的执行速度慢意味着会浪费用户的时间。软件的目的是实现业务的高效化，为用户节省更多的时间。代码执行速度慢的话就违背了这一目的。

  况且，代码执行速度慢还会引来用户的投诉。这时，我们不仅要花时间应对用户投诉，还会失去用户的信任。

- **代码可读性高 / 代码可读性低**

  有些人写出的代码可读性高，有些人写出的代码可读性低。

  另外，有些人写出的代码便于修改，有些人写出的代码一经修改就会出问题。

  二者由此产生的优化成本大不相同。代码质量差到一定程度时甚至无法优化。

综合上面几点来看，有能力的程序员和没能力的程序员确实差出好几个档次。

有能力的程序员在项目中起到的作用非常大。对于这些有能力的程序员，我们不可以将他们视为可交换的“1 人月”，要把他们留在项目中承担固定的职责。

关联信息 反模式

从成功的软件或软件开发中找出成功的原因，也就是找出良性的（共同）模式，对其进行分类并共享，这在当今已经是很常见的做法了。利用成功的模式，我们就可以在成功事例的引导下，大幅节约时间、费用与劳力。

人们在各个层级收集了大量模式。其中较具代表性的有代码设计中的“设计模式”、架构中的“企业应用架构模式”和团队编排中的“组织模式”等。

另一方面，人们也开始从失败的软件或软件开发中寻找失败的原因，也就是找出恶性的共同模式，对其进行分类并共享。为了与良性模式进行区分，我们称这类模式为“反模式”。本章就总结了各类反模式。

反模式用作“反面教材”，也用作“拐杖”。反模式是“陷阱”的定义集。了解了反模式，我们就能提早发现问题的苗头，防患于未然。在已经出现问题的情况下，只要能及时确认，就可以找到合适的解决方案。

出处

[1] 弗雷德里克・布鲁克斯. 人月神话 [M]. 汪颖，译. 北京：清华大学出版社，2002.

相关图书

[1] 大槻繁. ソフトウェア開発はなぜ難しいのか～「人月の神話」を超えて [M]. 東京：技術評論社，2009.

[2] 罗伯特・格拉斯. 软件工程的事实与谬误 [M]. 严亚军，龚波，译. 北京：中国电力出版社，2006.

[3] 史蒂夫・迈克康奈尔. 代码大全（第 2 版）[M]. 金戈，汤凌，陈硕，张菲，译. 北京：电子工业出版社，2006.

[4] 威廉・ J. 布朗，等. 项目管理反模式诊断 [M]. 杨晓燕，任树芳，王虹，等，译. 北京：电子工业出版社，2004.



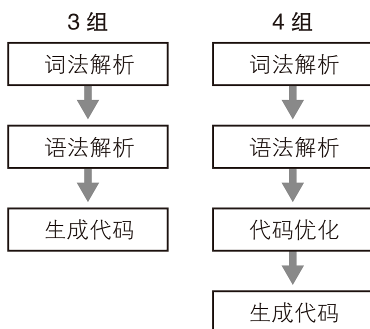
康威定律

英　语 Conway's law

是什么　架构服从于组织结构

软件的结构，即架构，反映了创建它的组织的结构。

比如以 3 组结构创建编译器，我们就能得到 Three-Pass 编译器；以 4 组结构创建编译器，我们就能得到 Four-Pass 编译器。



为什么　架构以信息交流为基础

设计软件的组织结构倾向于在设计中模仿软件的信息传递结构，因为信息交流建立在组织结构的基础之上，而架构建立在信息交流的基础之上。

可以说，软件的架构是由组织结构的通信路径形成的。

怎么做　先设计架构再编排组织结构

在什么都不做的情况下，架构是服从于组织结构的。但正确的做法恰好相反。也就是说，我们应该先设计一个好的架构，然后根据架构编排组织结构。

依照组织结构设计出来的架构，从软件的观点来看不可能是最优的。这种架构会给产品的质量带来负面影响。

但是，如果不考虑组织结构，只将架构调整到最优状态，通信路径又会发生错位。因为这时组织结构的组成要素不能反映软件的组成要素，组织结构间的关系也可能无法反映软件各个组成要素间的关系。在这种情况下，开发不可能按照预期进行。因此，应该先构建架构，然后让组织结构从属于架构。

不过，早期创建出来的架构只是一个半成品，此时的架构并不稳定。如果过早让组织结构从属于架构，组织结构就会跟不上架构的变化。

所以，等架构得到充分验证之后再构建组织结构的做法比较高效。

扩展　组织结构与过程

组织结构与架构之间相互依赖的关系固然重要，但我们也不能忽略组织结构与过程的契合度。

在数据库组、主框架组、Web 服务器组和测试组这种以技术领域分组的组织结构中实践敏捷过程，显然比较困难。

因为这些根据技术分出来的组，不论实现什么功能，都会过分依赖其他组的工作。即便是一个很小的功能，组与组之间也要进行大量的交流与合作，既消耗时间，又影响质量，还会增加成本。

在这种情况下，即使采用了敏捷开发，也不能一点一点地进行发布。可见，组织结构、架构和过程是紧密相连的，我们不能单独拿出某一项来评判好与坏。

出处

[1] James O. Coplien, Neil B. Harriosn. Organizational Patterns of Agile Software Development[M]. Upper Saddle River, New Jersey: Prentice Hall, 2005.

相关图书

[1] Maryand Tom Poppendieck. Leading Lean Software Development[M]. Boston: Addison-Wesley, 2009.



7.3 破窗效应

英 语 Broken Windows Theory

是什么 不好的代码是“蚁穴”

如果大楼这类建筑物上有一扇长期未被修理的窗户，这栋大楼就会给人一种“被遗弃”的感觉。人们便不会再留心这栋大楼的状态。

这样的话，还会有窗户继续碎掉。接着是垃圾乱倒，满墙涂鸦。别看只是破了一扇窗户，如果放置不管，整栋建筑也会遭到严重的破坏。

软件也会发生这样的事情。如果对软件的“破窗”，也就是那些不好的设计、错误的决定或不好的代码放置不管，那么不论它多么微不足道，也能在很短的时间内让整个软件腐烂。

为什么 不好的代码会带来邪念

软件中一旦存在“破窗”，程序员的脑中就会不自觉地产生“剩下的代码肯定也是一团糟，随便改一改算了”的想法。

关于这种现象，有一个叫作“信箱实验”的著名心理学实验。如果自家信箱附近的墙壁上有涂鸦，或者信箱附近有垃圾，那么该信箱中信件被盗的概率就会达到 25%。仅仅是一些垃圾和涂鸦，就能将许多正直人士变成小偷。

除了从众心理之外，我们也可以用“莫名的不安”这种心理来解释为什么会出现这种现象。一扇被弃之不管的破窗户，会让人产生“在这附近遇到危险的话肯定没人来救”的想法，随之让人产生不安的情绪。即便是一些细枝末节的东西，如果总是以一种没有得到处理的状态摆在

人们眼前，也会让人渐渐变得神经质，使人的交感神经处于紧张状态，甚至促使人付诸暴力。

也就是说，出现这种现象的关键原因，与其说是“破窗户”本身，不如说是小小的问题被弃之不管而带来的“不安”。相较于时间短强度大的精神压力，人们对时间长强度小的精神压力更加敏感。当某些有违社会道德的现象一直出现在我们的眼前时，人就会暴露出脆弱性。

怎么做 保持代码整洁

我们不能对代码的“破窗”，也就是代码不好的部分放置不管，要在发现“破窗”的时候立即进行修补。没有了“破窗”，代码就能保持整洁的状态，这样一来，程序员便会小心翼翼地对待这些代码，避免弄脏它们。就算交付日期近在眼前，也没人愿意当第一个弄脏代码的人。

另外，如果没有足够的时间修复代码，至少要简单明了地指出“这段代码不好”。

比如对于自己认为不好的地方，可以添加带标签的注释以显示在 IDE（Integrated Development Environment，集成开发环境）的任务列表里。这么做的目的是强调这些不好的地方已经得到了管理，防止损害进一步扩大。

扩展 人会模仿人

破窗效应既与“莫名不安”的心理因素有关，也与“反射性模仿他人行为”的人类自身特性有关。

心理学中已经证实，人类在婴儿时期就已经具备“反射性模仿他人行为”的特性了。不过，这个特性需要有足够长的时间才会显现出来。如果人们长期处于一种低素质的“习惯性懈怠”的状态，就会去模仿他人不好的行为，最终陷入恶性循环，这也可能是破窗效应出现的原因。

不过，不管是因为“莫名不安”还是“反射性模仿他人行为”，及时解决不好的代码都是不变的应对策略。

出处

[1] 安德鲁・亨特，大卫・托马斯．程序员修炼之道 [M]．周爱民，译．北京：电子工业出版社，2011.

相关图书

[1] 植木理恵．フシギなくらい見えてくる！本当に分かる心理学 [M]．東京：日本実業出版社，2010.



熵增原理

英 语 Law of entropy increase

是什么 代码会自然而然地开始腐坏

熵是物理学术语，表示体系的混乱程度。根据热力学法则，人们证明了全宇宙的熵处于增加状态。

软件开发可以超越大部分的物理法则，却逃不出熵增原理的束缚。如果不对代码进行管理，其混乱程度就会不断加深，直到突破极限。也就是说，代码会逐渐转向腐坏。

为什么 代码会向着混乱的方向转变

代码变得越来越混乱是软件开发中自然而然的事情。

不管开头多么有序，只要过上一阵子，代码就会开始腐坏。就像生肉放久了会变质一样，随着时间的推移，代码的腐坏程度会越来越深。臃肿的代码越积越多，使得维护难度不断增大。用不了多久，即便是很小的修改都需要耗费大量劳力，迫使我们不得不重新设计软件。

在这种情况下，重新设计软件很难一帆风顺。如今的软件日新月异，新的设计必须能跟得上时代的变迁才行。

也就是说，我们就算有非常明确的目标，也难免会跟不上步调，因为我们在实际工作时打的是“移动的靶子”。

怎么做　抓住代码腐坏的征兆

代码开始腐坏时有几个征兆。不要放过这些征兆，发现它们后立刻处理。

- **刻板**

  刻板指不容易修改代码。

  仅仅因为一处修改，就需要对所有与其存在依赖关系的模块进行修改，我们称这种代码设计为刻板的设计。

  刻板的设计会给我们带来很多困扰。比如我们接到委托，要对代码做一个很简单的修改，于是简单调查了需要修改的地方，预估了工作量。然而在实际工作时，随着工作的推进，我们还是要对其他预想之外的地方进行修改。结果，工作量远远超出预估，我们只能在规模庞大的代码中追查需要修改的地方。

- **脆弱**

  脆弱指一处修改会对其他部分的代码造成很大损害。脆弱的代码甚至会损坏与其完全不相关的代码。因此，程序员在处理新问题时就可能会引发其他问题，这就使程序员陷入追着自己尾巴跑的状态。

  毫不夸张地说，脆弱的模块并不罕见。这类模块很容易辨认。那些需要经常修复的模块、常年出现在故障列表中的模块、程序员认为需要重新设计的模块，以及越修复质量越差的模块等就属于脆弱的模块。

- **可移植性差**

  可移植性差指软件难以移植到其他环境中。

  如果软件在任何环境下分离可运行部分和依赖环境的部分都会出现困难并伴随风险，我们就可以说该软件不具备可移植性。

● **难以掌控**

难以掌控指代码难以掌控和开发环境难以掌控。

代码难以掌控是指设计结构不具备灵活性。我们无法在保持设计结构的前提下轻松修改难以掌控的代码。相较于能保持设计结构的方法，使用投机取巧的方法更能轻松地完成修改。在代码难以掌控的状态下，做错事容易，做对事反而难。

而开发环境难以掌控常发生在开发环境效率低下的时候。比如，当编译需要花费大量时间时，即使我们知道已经无法保持设计结构了，还是会倾向于采用能避免大规模编译的修改方式。如果提交确认两三个文件需要花费好几个小时，我们就不会再思考保持设计结构的方法了，而是会寻找更节约时间的修改方式。

● **复杂**

复杂指不必要的元素过多。

当程序员预判规格说明书会发生变更，在代码中事先埋下应对机制时，就容易使代码变得复杂。这类做法总给人一种好的印象。很多人认为预见未来并提早做出准备就能保持代码的灵活性，防止今后苦于修改。

然而很遗憾，这样做只会带来相反的效果。为应对更多不测，我们会在代码中留下大量一次都用不上的结构。这会让代码变得复杂，变得难以理解。

● **重复**

重复指同样的代码出现多次。

在写文档时，复制粘贴是一个很好用的方法，但在编辑代码时，使用复制粘贴则会招来很严重的后果。在代码出现重复的情况下，修改软件将成为一项劳神费力的工作。如果在重复的部分发现故障，就需要修改代码中所有相同的部分。

况且，代码有时候看上去相同，但实际上有着细微的差别，这时修改方式就可能不同了。

如果这种看上去相同但存在细微差别的代码在软件中大量出现，就表示程序员没有做抽象化工作。如果能找出所有重复的部分，将其适当抽象化，消除重复，系统将更容易理解且更容易维护。

- **不透明**

不透明指代码难以理解。

代码有时候很难让人理解。而频繁修改的代码会随着时间的流逝越来越难以让人理解。

在刚写完代码时，代码对编码者本人来说是非常明了的，因为编码者沉浸于开发，熟悉该项目的每个地方。然而过一段时间再回过头来看，编码者就会觉得自己怎么能写出如此不堪的代码。

为了防止此类情况发生，编码者需要站在代码阅读者的立场思考，写出别人能够理解的代码。让别人来看自己写的代码是一个行之有效的方法。

扩展一　在敏捷开发中代码不容腐坏

敏捷开发乐于接受变化。

在敏捷开发中，我们几乎不给初期设计留时间，因为初期设计会随着时间的推移不断劣化。把时间花在这方面得不偿失。

另一方面，在敏捷开发中，我们会尽可能地让设计保持简洁，尽可能频繁地进行单元测试和验收测试。

这样一来，设计就能保持灵活性且易于变更。敏捷开发这一开发方式能利用自身的灵活性持续地对设计进行改善，所以在各个迭代的最后，我们能获得最符合该迭代要求的设计。

扩展二　营造不允许代码腐坏的团队文化

使软件变得混乱不堪的最大因素是项目活动中的心理学，也就是文化。

要防止代码腐坏，就需要时刻注意让代码保持简洁，不然会出现各种问题，比如设计过剩、解决问题的方案比问题本身还要棘手、不注意本质问题只关注罕见问题等，使代码变得复杂。

是否重视腐坏的代码取决于团队文化。如果只是头痛医头脚痛医脚，容许腐坏的代码存在，代码的混乱程度就会迅速加重。为了防止这一情况发生，作为团队的行动方针，应该鼓励团队成员在发现腐坏代码时立刻进行重构。这一方针是以代码质量为先的。要营造一种让人不得不这样做的氛围，集全员之力修改问题代码，不给代码留腐坏的机会。不可否认，这样会消耗额外的时间与成本，但如果能让代码规避腐坏的风险，这么做也是值得的。

出处

[1] 安德鲁・亨特，大卫・托马斯. 程序员修炼之道 [M]. 周爱民，译. 北京：电子工业出版社，2011.

相关图书

[1] 罗伯特・ C. 马丁. 敏捷软件开发：原则、模式与实践 [M]. 邓辉，译. 北京：清华大学出版社，2003.
[2] 鈴木炎. エントロピーをめぐる冒険初心者のための統計熱力学 [M]. 東京：講談社，2014.



7.5 80–10–10原则

英 语 80-10-10 rule

是什么 编程没有万能药

我们在用高水平的工具或语言开发软件时，可以在非常短的时间内实现用户 80% 的需求。而在剩下 20% 的需求中，有 10% 的需求需要通过一定努力才能实现，另 10% 则完全不可能实现。

因此，如果要 100% 满足用户的需求，开发就会陷入进退两难的境地。

如果此时已经开发一部分内容了，那么抛弃原有工具重新开发就显得不切实际。这时，我们就得放弃使用工具，用最笨拙的方式来满足某部分需求。

为什么 编程的问题领域太广

软件行业从 20 世纪 90 年代中期起，举整个行业之力花费十几年做了一场实验。实验内容是创造一款能够让能力平庸的技术人员的生产效率飞跃性提升的“万能工具”，比如模型驱动开发、4GL（第四代语言）等。

实验的结果显示，使用单一工具很难在所有领域都获得完美的成果。

人们创建这种工具是为了开发出更人性化、质量更好的软件。因此，为了防止能力平庸的技术人员引发问题，人们对语言施加了相当强的功能限制。结果，工具产生了自己的“防守范围”。

但软件要处理的问题范围是无限大的。用一个工具解决所有问题的“万能药”路线显然走不通。

怎么做　工具要用在合适的地方

要记得，软件开发没有万能药。就算他人鼓吹某个工具如何如何好，我们也要知道世上没有完美的东西。

但是，精确到某个领域之后，我们确实能找到一些各方面能力都很均衡的高水平工具。这些工具能帮我们完成很多工作，并通过插件的方式代替原来的一些笨办法。这类工具只要运用得当，就能发挥很大的效果。

因此，我们应该积极导入工具。不过，盲目相信工具并不可取，因为这么做会影响效率。

导入工具前要先进行模拟开发，确认工具的适用范围及效果之后，再决定是否正式导入。

扩展　对症药比万能药好用

软件开发中没有能称为万能药的工具。但是，我们可以根据项目的特征制作对症药。换句话说，就是通过功能强大的语言 + DSL 的元层来创造专用语言。在这个领域中，使用的语言最好是动态的且易于元编程的语言。

关联信息　80:20 原则

有一个原则与 80-10-10 原则的名字很像，那就是 80:20 原则（帕雷托法则）。二者名字类似，但内容大不相同。

80:20 原则指整体的 20% 的元素创造了整体的 80% 的内容。这是一种在自然界和人类社会中常见的现象，比如：

- 20%的商品创造了80%的销售额
- 20%的道路上集中了80%的交通量
- 占人口总数20%的富裕阶层拥有80%的社会财富

- 20%的功能占用了80%的使用时间

编程中也有符合该原则的现象，比如：

- 20%的代码集中了80%的故障
- 20%的代码消耗了80%的处理时间

确实，代码存在质量及性能的“热点”。找到这些热点至关重要。为了方便说明，我们用了 20% 这个数值，但实际上这个数值更小。

故障集中的地方是质量的热点。因此，在发现故障时，应审查故障周围的代码，编写单元测试，进行水平扩展测试，集中解决问题。

另外，需要花费大量处理时间的地方是性能的热点。我们应通过测试检查出这些地方，先从这些地方开始进行性能调优。

出处

[1] 尼尔·福特. 卓有成效的程序员 [M]. 熊节，译. 北京：机械工业出版社，2009.



约书亚树原则

英 语 Joshua Tree Principle

简 写 JTP

是什么 没有名字的东西“不可见”

曾经有这么一个人，常年住在镇子上的他某一天在图书馆里阅读植物图鉴时，认识了一种名叫“约书亚树”的植物。图鉴中附有该植物的照片，而他从未在镇上见过这种植物。

然而，在离开图书馆回家的路上，他发现镇子里到处都是约书亚树——那种他以为自己从没见过的树。

这个故事告诉我们，人在知道某个东西的名字后就会注意到这个东西。反过来说，如果某个东西没有名字（我们不知道名字），那么人就很难注意到它。

为什么 知道名字才知道其存在

事物或概念以有名字（人们知道它的名字）为前提。有了名字才能让人们知道它，才能将信息传达给别人。

对肉眼看不到的概念类事物而言，名字尤其重要。这是因为有了名字之后，这类事物才能被认知，才能被人重复使用。

比如设计模式的最大成果就是给已有的设计窍门起了一个名字，供人重复使用。

怎么做 使用通用语言

我们要创造语言，然后在团队中共享。

这就需要用到通用语言（Ubiquitous Language，UL）了。通用语言是能够正确表达某问题领域各个要素的团队共有的语言。因为是共有的语言，所以在团队内必须实现统一。指代同一个东西时要用同一个词语。近似的词语会招来混乱，我们不允许其出现。

在没有通用语言的情况下，团队成员使用的语言会各不相同。如果每个人使用的是自己定义的语言，语言之间就会产生障碍，有时候甚至需要翻译。这样一来，交流上就需要花费更多的时间，意思的传达也将变得模糊。

因此，为了让团队凝聚成一体，我们需要让通用语言来充当糨糊的角色。在能正确捕捉问题领域的高价值软件设计中，通用语言是必不可少的要素。

通用语言不仅要用在平时的对话中，还要用到代码里。这样一来，说明文档和代码都能实现统一，都变得易于理解。

另外，我们要反复琢磨通用语言。读出声，或者落实在笔头，多找一些候补选项，并从中提炼出合适的概念与词汇。

关联信息 巴别塔

巴别塔是《圣经·旧约》的《创世纪》中出现的巨塔。人类想建设通天的高塔，却遭到了神的阻止，最终失败。根据该典故，后人用“巴别塔”来讽刺那些不切实际的计划。

神阻止人类建设巴别塔的方法是打乱人类的语言。神降临人间，见到巴别塔后说：“人类就是因为有通用的语言才做出这等事情。应打乱人类的语言，让他们听不懂对方说的话。”神真的这么做了。结果，语言被打乱的人类陷入混乱，被迫终止塔的建设，分散到世界各地。就这样，建设巴别塔的项目失败了。

可见，要想让项目成功，通用的语言必不可少。

出处

[1] 罗宾・威廉姆斯. 写给大家看的设计书 [M]. 苏金国，刘亮，译. 北京：人民邮电出版社，2009.

相关图书

[1] 埃里克・埃文斯. 领域驱动设计 [M]. 赵俐，盛海艳，刘霞，译. 北京：人民邮电出版社，2010.
[2] 橋爪大三郎. 教養としての聖書 [M]. 東京：光文社，2015.



第二系统综合征

英 语 Second system syndrome

是什么 第二次发布总会出现功能过多的情况

由发布第一版软件的程序员设计的第二版软件会成为最危险的一个版本。

第二版软件有功能过多、质量差以及功能的使用体验较差等倾向。

为什么 人在适应开发后会倾向于“多功能主义”

在开发第一版软件时，由于未知的情况很多，风险较高，所以我们在进行判断时会比较慎重。即便想到了好的功能，也会留到下一次再实现。

然而，在开发第二版软件时，我们掌握了更多的信息，也有了自信，所以倾向于把之前保留的功能以及新想到的功能一股脑儿加进去。

添加过多功能之后，代码变得复杂，不易维护。功能本身也变得复杂，使用体验变差，结果添加的功能也没能得到人们的青睐。不管是代码还是实现的功能，质量都较以前有所下降。

另外，那些暂时保留的功能在第一版软件中也许是比较实用的，但在第二版软件中，这些功能可能已经失去了必要性，或者落后于时代了。也就是说，把这部分功能放到第二版软件中实现是一种浪费时间的做法。

怎么做 考虑用户

程序员要有自制力，避免陷入多功能主义的怪圈。

要做到这一点，一个有效的做法就是重新对用户进行定义并将用户具象化。此时不论是有意识的还是无意识的，程序员对用户的印象都会对程序员的判断产生影响。这就给程序员添加新功能的欲望带上了“枷锁”。

具体做法就是在编程时多想想以下问题。

- 用户是谁
- 用户需要什么
- 用户认为什么是必要的
- 用户想要什么

扩展 第二系统后综合征

前面说程序员容易在第二版软件中产生多功能主义的倾向，但实际上，第二版以后的版本也会出现同样的情况。

特别是数据包软件等需要持续发布的软件，随着一次次版本升级，没用的功能会越来越多。

出现这种现象的原因可能是用户群体不固定，程序员很难对用户进行具象化。而且功能一旦发布就很难有机会删除，所以只能越积越多。

不可否认，添加功能可以提升软件的魅力。但是，相较于新功能，用户往往希望基本功能是稳定的，或者基本功能的使用体验能得到改善。

关联信息 功能蔓延

功能的过分扩张不能全部归罪于程序员的一己私欲，毫无原则地满足用户的需求也是重要原因之一。

无条件满足用户的愿望，就会在软件中增加大多数用户用不到的功能，还要准备用于控制该功能的复杂的设置画面以及相关设置文件。如此一来，软件就会变得难以维护，故障频出。

这种功能肆意增多的现象称为功能蔓延（feature creep），该现象意味着软件开始迈向破灭（或者已经破灭了）。

软件设计的终极之美是“简单”。越是简单优质且拥有众多用户的软件，越容易出现更多的需求。如果忠实地满足这些需求，将所有功能都开发出来，软件将失去简单性，变成一款没人用的软件。这时我们就会陷入进退两难的窘境。

避免出现这种悲剧的关键是要有勇气对需求说“NO”。对于那些与软件核心无关、需要与其他软件组合才能实现的功能，我们要明确地说“NO”。只有这样，才能产生优秀的设计，才能让软件保持简单性。

不过，有时候我们很难拒绝用户强烈的诉求。这时，我们不要直接在软件主体中实现该功能，而是要围绕软件主体进行扩展，或者以插件的形式在不改变软件核心代码的前提下修改软件的运行模式，以此来保持软件主体的简单性。

出处

[1] 弗雷德里克·布鲁克斯. 人月神话 [M]. 汪颖，译. 北京：清华大学出版社，2002.

相关图书

[1] 凯佛林·亨尼. 程序员应该知道的 97 件事 [M]. 李军，译. 北京：电子工业出版社，2010.



重新发明车轮

英 语 Reinvented wheel

Reinventing the wheel

是什么 制作已有的东西

有时候对于某种功能，明明有现成的代码或库可以使用，人们却还要自己重新开发相同的功能。这就像专门花时间又重新发明一遍世上早就有的车轮一样，是一种无用功。

有现成的东西，却要去重新发明一个，这是在浪费时间。当开发规模足够大时，其危害也是非常大的。想要一个“能运行各种服务的服务器”，于是专门把 Web 服务器这种规模极大的软件重新开发了一遍。这种做法会浪费非常多的时间。

而且在大部分情况下，相较于重新发明出来的东西，既有产品的质量更好。比如相较于我们现写出来的库，既有的标准库更好，因为它不仅能反映出提供标准库的专家的知识，还能反映出人们在使用过程中积累的经验。标准库还有一个好处，那就是就算我们什么都不做，随着时间的推移，其中的故障、功能和性能也会自行改善。

另外，如果忽视标准规格，根据自己的协议编写代码，将来就只能走自己的路线了。仅靠本地的几个程序员是不可能跟得上世间的主流的。另外，由于所有的插口都是独创的，所以将来也无法实现替换。

为什么 不知道车轮和想制作车轮

重新发明车轮的原因有以下两种。

● **不知道车轮**

程序员不知道车轮的存在。也就是说，这种发明不是程序员有意而为的。

这归咎于程序员的知识不足和学习不足。编写与语言标准库功能相同的代码，或者在有标准协议的情况下用独创的格式编写通信功能的代码等都属于这种情况。

● **想制作车轮**

程序员有制作车轮的欲望。也就是说，这种发明是程序员有意而为的。

这是一种叫作“非我发明”（Not Invented Here，NIH）综合征的问题。具体表现为某个东西原本没有重新制作的必要，程序员却出于对技术的兴趣或排斥他人制作的东西而想重新制作一遍。

怎么做 关注车轮之外的东西

我们要避免重新发明车轮，将重点放在本来应该做的工作上。

为此，在编写代码之前，一定要先确认是否存在相同功能的标准库、开源库，是否存在标准协议等。

另外，要借助团队会议等机会从其他程序员处获取信息。这样就能避免团队内出现重复劳动的情况。

同时，在团队中彻底清除利己主义的思想。

因为想做而做，这是程序员自私的一面。然而，软件的目的不是满足程序员的欲望，而是满足用户的需求。为了用户，为了在质量、开发时长和费用等方面做到最好，我们应该时常调查哪些东西可供使用，掌握高质量的开源工具或商用工具。

扩展 允许重新发明车轮的情况

有时我们也需要大胆地重新发明车轮。

- **商业目的**

商业上的核心部分必须由自己制作。

在使用已有的东西时，必然会对该部分产生依赖。依赖则意味着对该部分失去了控制权。

即便知道其中潜藏着致命的问题，我们也无法主动去修改。就算可以委托他人修复，何时能够发布，是否真的能得到改善，都是未知数。质量和交付期方面的问题很可能在商业上造成无可挽回的损害。

况且，使用已有的东西就意味着放弃该部分的“差别化”。因此，商业上的核心部分，从原则上来讲都应该由自己制作。只有自己制作出这部分内容，并且花心思做出个性，从中积累经验，才能开发出独特的、能贡献于世界的软件。

- **学习目的**

要成为优秀的程序员，就得不断积累高质量的经验。

软件开发的模式、设计和编程等方面的好书有很多，然而读书和实践之间有很大的差别。

同样，借用已有的代码与自己从零设计、测试软件，解决故障，提高软件质量得来的经验有天壤之别。

不过，有机会编写软件核心部分代码的程序员少之又少。大部分程序员只能借用已有代码。在这种情况下，我们不知道代码内部是如何运作的，因此和使用“黑箱”没什么区别。

只看水面的话，我们是无法得知水下隐藏着何种危险的。如果不知道水底究竟发生了什么，就不能灵活运用水流。自己亲手制作是一种必要的经历。为此而“重新发明车轮”是程序员学习、提高技术非常有效的一个方法。

当然，我们免不了失败，但这种经历也比直接拿现成的使用要宝贵。亲手从零开始写代码，进行各种尝试，从一次次失败中学习，能带来不同于阅读技术类图书的好处。不过，读书与实践同等重要，它们对程序员来说都是不可或缺的。

出处

[1] 芭比・戴维斯. 项目经理应该知道的 97 件事 [M]. 张科, 焦亚超, 译. 北京: 人民邮电出版社, 2011.

相关图书

[1] 约书亚・布洛克. Effective Java 中文版（第 3 版）[M]. 俞黎敏, 译. 北京: 机械工业出版社, 2018.

[2] 凯佛林・亨尼. 程序员应该知道的 97 件事 [M]. 李军, 译. 北京: 电子工业出版社, 2010.

[3] 贾里德・理查森, 威廉・格沃特尼. 软件项目成功之道 [M]. 苏金国, 王少轩, 译. 北京: 人民邮电出版社, 2011.

[4] 卡纳特・亚历山大. 简约之美: 软件设计之道 [M]. 余晟, 译. 北京: 人民邮电出版社, 2013.

[5] 查德・福勒. 我编程, 我快乐 [M]. 于梦瑄, 译. 北京: 人民邮电出版社, 2010.

[6] William J. Brown, Raphael C. Malveau, Hays W. "Skip" McCormick, Thomas J. Mowbray. AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis[M]. New York: Wiley, 2008.

[7] 乔尔・斯波尔斯基. 软件随想录 [M]. 杨帆, 阮一峰, 译. 北京: 人民邮电出版社, 2015.



7.9 给牦牛剃毛

英 语 Yak Shaving

是什么 抓不住问题的本质

有种家畜叫牦牛。它是牛的一种，特征是身上长着厚厚的毛。每当临近夏天，牦牛就需要剃毛。我们需要给牦牛剃去相当多的毛才能让它的皮肤露出来。

我们处理某些问题时就像给牦牛剃毛一样，在解决问题的过程中总会有新的问题冒出来，让我们难以抓住问题的本质。

这种状态如果持续太久，人们就可能会忘记原本要解决的问题是什么。

为什么 问题会接二连三地出现

问题总是接二连三地出现。

假设我们想导入在 Web 服务器上运行的任务自动化工具，以提高工作效率。

“先下载Web服务器程序。”
“文件太大了，没有办法下载。”
“那就导入下载工具。”
“下载工具怎么不运行呀？”
“原来需要前置模块啊。”
“那就下载前置模块。”

“需要注册用户。”
“那就注册一个吧。”
“诶？用户注册页面不动了。”
“原来是浏览器版本太老了。”
“升级了浏览器，注册了用户，模块也下载好了。”
“怎么下载工具还是不运行？”
“哎呀，需要操作系统的补丁包。”
（后面依然没完没了。）

这种像给牦牛剃毛一样的情况会造成时间上的浪费。有时候，就算我们预估了工作所需时间，也没有办法在预估的时间内完成工作，这种情况发生的原因就是我们把时间耗费在了给牦牛剃毛上。

另外，在给牦牛剃毛的状态下，人非常容易积攒压力。我们很难推测出需要花多长时间才能把牦牛身上的长毛剃光。如果这种无法达成目标的状态一直持续下去，人就会产生挫败感。

怎么做 尽早收手

当我们发觉自己已经陷入给牦牛剃毛的状态时，应停下脚步，回想自己原本要实现的目标是什么。如果发现自己已经偏离了目标，或者从时间、成本的角度来看不适合再继续操作下去了，应立刻停止工作。因为在这种情况下，寻找其他出路往往会带来更好的结果。

另外，为防止其他人也陷入同样的状态，我们应将整个过程分享给团队成员。在一个全员共享的空间留下一份笔记，能够防止他人浪费时间。

扩展一 勇于面对“给牦牛剃毛”

一般来说，见到要给牦牛剃毛的情况应该绕着走。但是，出于一些有价值的目的，或者因为紧急故障等，有时我们必须跨越“给牦牛剃毛”的障碍，解决问题。

这个时候最麻烦的是我们大脑解决问题的速度跟不上问题出现的速度。由于前一个问题尚未解决就冒出了下一个问题，所以我们的大脑在解决问题时往往像使用栈一样，先让问题入栈，再一个一个出栈解决（同时让继发的新问题入栈）。这就是给牦牛剃毛的状态。在这种情况下，问题通常会接二连三地发生，出栈速度赶不上入栈速度，导致脑内栈溢出。

为防止这类情况发生，我们要记住不能只在脑中解决问题。应当把问题写下来，一个一个地解决。

扩展二　编程中的“给牦牛剃毛”

给牦牛剃毛的情况常出现在搭建环境的过程中。不过，编程中也会遇到类似的情况。

比如，在写代码时，由一个问题联想到其他问题，离最初要解决的问题越来越远。在最坏的情况下，我们甚至会忘记最初或中间想到的问题是什么。

又比如，在读代码时，由于代码未整理，所以我们很难找到当初想知道的东西。在梳理错综复杂的调用关系时，一不小心就会忘记代码读到了哪里，或者读代码的目的是什么。

为防止这类情况发生，我们在读写复杂的代码时，要一边做记录一边操作。特别是在写代码时，我们需要思考的部分比实际操作的部分要多，不做记录的话就可能会有陷入循环思考的状态。

出处

[1] 尼尔・福特. 卓有成效的程序员 [M]. 熊节，译. 北京：机械工业出版社，2009.

后　记

在本书的最后，笔者将主题从“编程”上升到“做人”，为大家介绍三个原则。

* * *

第一个原则是“关怀”，出自现代哲学家梅洛夫的著作《关怀的力量》。

关怀从字面意思上讲，就是关心他人。

关心他人是指帮助对方成长并实现自我，但关怀的意义不止于此。除了帮助他人成长之外，帮助自己成长也是关怀的意义之一。帮助他人实现自我有助于自身的自我实现。

注意，不是为了让自己实现自我而去帮助他人，而是帮助他人成长可以完成自身的自我实现。

关怀的对象不一定是人。比如艺术家就可以把作品当成关怀的对象。

对程序员而言，关怀的对象是代码和读代码的人。程序员要精心编写代码，要写出能让人轻松读懂的代码。

“易懂”属性能帮助代码成长，同时给读代码的人带来方便。

这些关怀最终会回报到自己身上。

* * *

第二个原则是“道德法则”，出自近代哲学家康德的著作《实践理性批判》。

道德法则是制约人类道德行为的普遍标准。这一基准以“你这样行动，使你意志的准则始终能够同时成为普遍立法的原则”的命题形式来表现。

说得通俗一点，就是采取行动前先问一下自己这个行动能否获得众人认可，如果能，则立即执行。

该法则延伸出了很多法则，这些法则都有一个核心的内容，那就是人要永远把自己人格中与他人人格中共有的人性作为目的，决不可单独将其用作手段。

不论是自己还是他人，人这种生物，都拥有无可替代的人格。人格不应作为手段来使用。人和人之间也不可以将对方的人格视为手段。我们要把自己以及对方的人格看作目的。

程序员必须记得，软件的用户和自己一样都是拥有人格的人类。用户与程序员的关系不是互相利用，而是互相为对方做出贡献。

用户有用户的目的，软件就是为了达到用户的目的而开发的。程序员应在用户以及用户使用软件的方法上多花心思，开发出能满足用户需求的有用的软件。

* * *

第三个原则是“中庸”，出自古代哲学家亚里士多德的著作《尼各马可伦理学》。

“中庸”是调节人类行为及感情中过剩部分与不足部分的“德”。这个“德”指对待任何事情都不走极端，以恰到好处的平衡之道来应对各种情况。

比如，将中庸之道用在应对恐怖情况上时就是“拿出勇气”。超过中庸这条线就是“蛮勇”，没有达到这条线则是“怯懦”。

在软件开发现场，很少有答案非此即彼的问题。存在标准答案的问题少之又少。解决方案的选项都是“渐变色”。

黑与白之间存在着无数种浓淡不同的灰色，因此不存在黑白两端二选一的情况。

我们要认识到解决方案的选项是“渐变色”的事实，并以“选择哪种浓度的灰色”的思考方式来对待问题。

为此，掌握“语境”就显得至关重要。我们要以从语境中得来的信息为基础，找准平衡点，选择最合适的“灰色”。

* * *

以上就是三个做人方面的原则。

程序员写出来的代码会有人读，开发出的成品软件也会有人用。在

编写代码时，应照顾到这些人，掌握语境，做出最为平衡的选择。

到此，所有原则就介绍完了。

自然科学成果展示了世间万物不过是一种“信息状态”。如果是这样，那么负责处理信息的程序员可以说身担重任。这是一个有无限可能且十分有意义的职业。

衷心希望本书能给各位程序员带来启发，帮助大家写出更好的代码，开发出更好的软件。同时希望本书能给整个社会带来好的影响。

上田勋
2016 年 1 月

出处

[1] Milton Mayeroff. On Caring[M]. New York: William Morrow Paperbacks, 1990.

[2] 御子柴善之. 自分で考える勇気——カント哲学入門 [M]. 東京: 岩波書店, 2015.

[3] 亚里士多德. 尼各马可伦理学 [M]. 邓安庆，译. 北京: 人民出版社，2010.

谢 辞

首先，衷心感谢秀和系统编辑部的责任编辑，使原本杂乱无章的信息升华为有益于读者的文章，最终成就了本书的出版。

感谢诸多技术图书和技术杂志的出版方。鄙人的所有知识全来自于各出版方出版的图书和杂志。特别是其中介绍的各个原则让我受益匪浅，我也因此而成长许多。

感谢在网络上公开技术信息的各位。如果没有好书介绍、学习方法介绍等信息，我恐怕无法看清前路，至今仍感到迷茫。另外，若不是各位提供了关于编程的具体信息，我也不可能写出代码。

感谢阅读鄙人博客的各位。大家来读我的博客是对我的鼓励，我也因此维持了较高的学习效率。

感谢各位团队成员一直以来对我的帮助。能够开发出好的产品，以及能让本书有如此丰富的内容，都是各位的功劳。

感谢生我养我的父母。正因为有你们，我才有今天。

最后致我的妻子。

虽然你是编程方面的外行人，但还是为我的文章做了校对工作，谢谢你。你准确指出了我笔误的地方以及文章中的一些漏洞，给了我很大的帮助。有君相佐乃此生之幸。

有幸降生在这个物理世界，委实是一件值得感谢的事。

余生也望多多指教。

版 权 声 明



微信连接



回复“53914”获取本书配套资源



回复“修炼”查看相关书单



微博连接

关注@图灵教育 每日分享IT好书

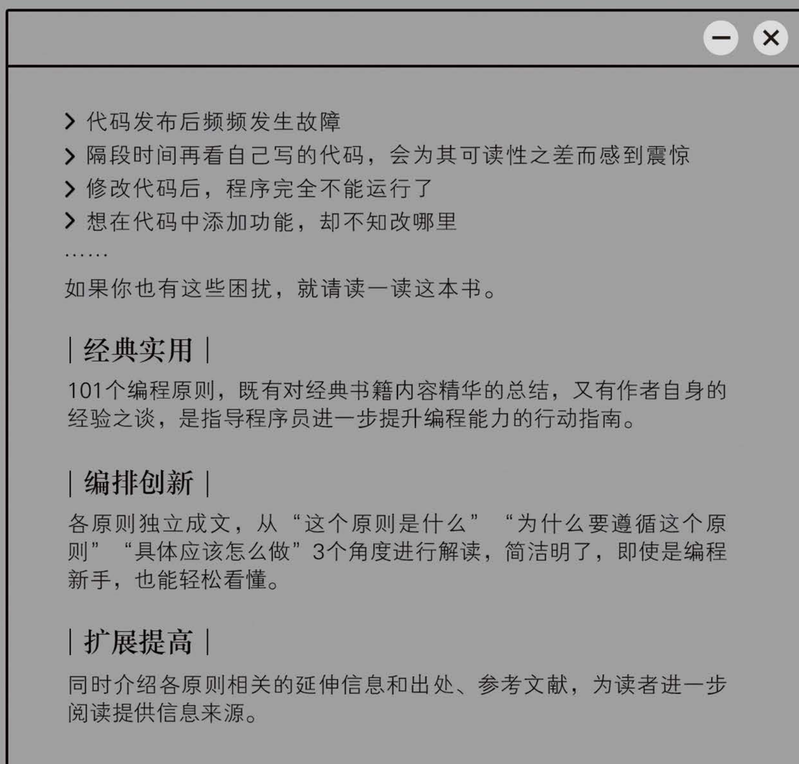


QQ连接

图灵读者官方群I：218139230
图灵读者官方群II：164939616

图灵社区
iTuring.cn

在线出版，电子书，《码农》杂志，图灵访谈



- 代码发布后频频发生故障
- 隔段时间再看自己写的代码，会为其可读性之差而感到震惊
- 修改代码后，程序完全不能运行了
- 想在代码中添加功能，却不知改哪里

……

如果你也有这些困扰，就请读一读这本书。

| 经典实用 |

101个编程原则，既有对经典书籍内容精华的总结，又有作者自身的经验之谈，是指导程序员进一步提升编程能力的行动指南。

| 编排创新 |

各原则独立成文，从“这个原则是什么”“为什么要遵循这个原则”“具体应该怎么做”3个角度进行解读，简洁明了，即使是编程新手，也能轻松看懂。

| 扩展提高 |

同时介绍各原则相关的延伸信息和出处、参考文献，为读者进一步阅读提供信息来源。

图灵社区：iTuring.cn
分类建议：计算机/软件开发
人民邮电出版社网址：www.ptpress.com.cn

